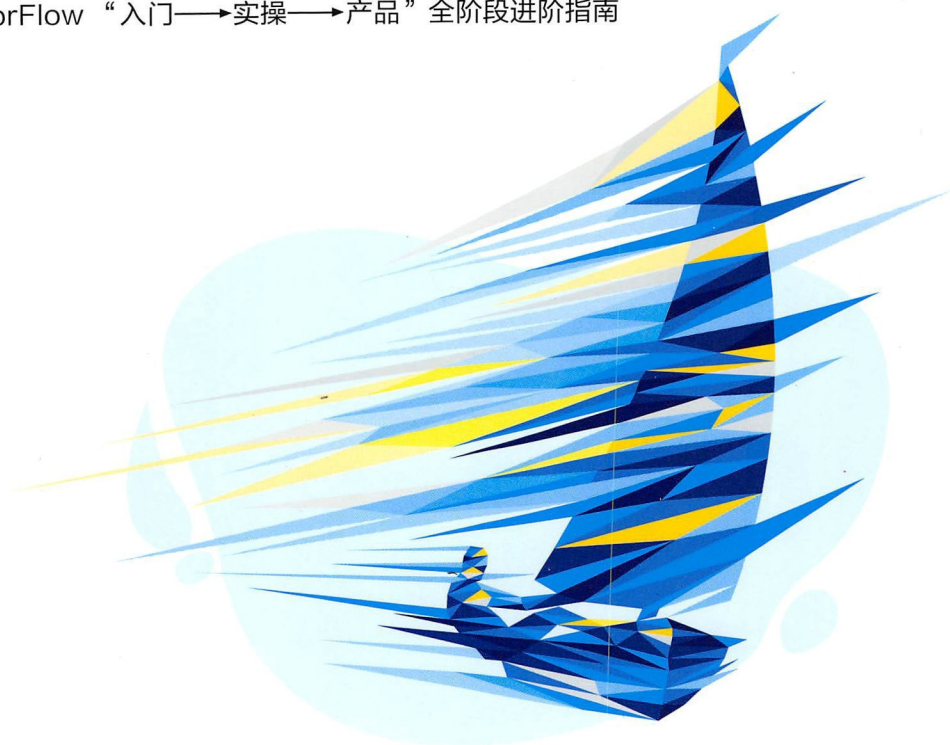


版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



TensorFlow

进阶指南

基础、算法与应用

黄鸿波◎编著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn



作者简介



黄鸿波

珠海金山办公软件有限公司（WPS）人工智能领域专家，高级算法工程师，拥有多年软件开发经验。

曾在格力电器股份有限公司大数据中心担任人工智能领域专家。曾带领团队开发过基于人脸识别技术的智能支付系统、推荐系统、知识图谱、智能问答系统等。擅长数据挖掘、机器学习、移动开发等，并拥有丰富的实战经验。

本书服务邮箱：
service@tensorflow-guide.com



博文视点AI系列



TensorFlow

进阶指南

基础、算法与应用

黄鸿波◎编著

电子工业出版社
Publishing House of Electronics Industry
北京·BEIJING



内 容 简 介

本书是由人工智能一线从业专家根据自己日常工作的体会与经验总结而成的，在对 TensorFlow 的基础知识、环境搭建、神经网络、常用技术的详细讲解当中穿插了自己实战的经验与教训。更与众不同的是，本书详细地解析了使用 TensorFlow 进行深度学习领域中常用模型的搭建、调参和部署整个流程，以及数据集的使用方法，能够帮助您快速理解和掌握 TensorFlow 相关技术，最后还用实战项目帮助您快速地学会 TensorFlow 开发，并使用 TensorFlow 技术来解决问题。

本书代码主要是在 1.6 版本的基础上进行开发的，同时兼容 1.2~1.10 的版本，并已得到验证。本书主要面向对 TensorFlow、深度学习、人工智能具有强烈兴趣且希望尽快入门的相关从业人员、高校相关专业的教育工作者和在校学生，以及正在从事深度学习工作且希望深入的数据科学家、软件工程师、大数据平台工程师、项目管理者等。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目(CIP)数据

TensorFlow 进阶指南：基础、算法与应用 / 黄鸿波编著. —北京：电子工业出版社，2018.11
(博文视点 AI 系列)

ISBN 978-7-121-34565-4

I. ①T… II. ①黄… III. ①人工智能—算法—指南 IV. ①TP18-62

中国版本图书馆 CIP 数据核字(2018)第 135156 号

责任编辑：孙学瑛

印 刷：三河市双峰印刷装订有限公司

装 订：三河市双峰印刷装订有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787×980 1/16 印张：23.25 字数：446 千字

版 次：2018 年 11 月第 1 版

印 次：2018 年 11 月第 1 次印刷

定 价：99.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888, 88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819, faq@phei.com.cn。



推荐序

这是一个最好的时代，这是一个最坏的时代；
这是一个智慧的年代，这是一个愚蠢的年代；
这是一个信仰的时期，这是一个怀疑的时期；
这是一个光明的季节，这是一个黑暗的季节；
这是希望之春，这是失望之冬；
人们面前应有尽有，人们面前一无所有；
人们正踏上天堂之路，人们正走向地狱之门。

这是英国著名作家查尔斯·狄更斯在《双城记》中的名句，除了最后一句以外，形容现在这个时代再合适不过了。电子技术、计算机技术、互联网技术、大数据技术、人工智能技术，这些技术的发展与进步是一股不可阻挡的历史洪流，而我们都是洪流中的小舟。

乐观与悲观，兴起与沉沦，都在人们的一念之间。任何一种技术的革命都伴随着一代人的分化与沉浮。

但我相信，所有不愿辜负时代的智者和勇者都会以最积极的方式面对大潮的到来，因为迄今为止没有任何一个时期会比现在这如井喷一般繁荣绚烂的新技术所带来的
人工智能新纪元还令人神往。

抓住它，就是抓住航船的舵柄；抓住它，就是抓住未来的航标。

让我们乘着这时代的大潮，一起破浪远航吧！

姚冬

金山软件集团副总裁



前言

自从 2016 年初谷歌的 AlphaGo 战胜了世界围棋冠军李世石，人工智能和机器学习领域就变得异常火爆。其实早在半个世纪以前，人工智能领域的相关概念就已经被提出，经过了这半个世纪的不断发展，人工智能以及深度学习相关领域已经在工业界和学术界发生了颠覆性的改变。2017 年 2 月 11 日，谷歌公司正式发布 TensorFlow 1.0 版本，由此开始，一段新的热潮被掀起。到目前为止，谷歌公司的 TensorFlow 版本已经更新至 1.10 版本，本书代码主要是在 1.6 版本的基础上进行开发的，同时兼容 1.2~1.10 的版本，并已经过验证，读者可以放心使用。

关于本书的使用

本书主要面向对 TensorFlow 具有强烈兴趣且希望尽快入门的读者，以及正在从事 TensorFlow 方面的工作且希望深入的读者。

本书一共有 14 章的内容，分为四个部分：

TensorFlow 基础部分（第 1 章到第 3 章）

主要讲解了 TensorFlow 的基础知识（包括张量、运行图、Session 等部分）、环境的搭建方法（包括 Windows、Linux 版本的环境搭建，以及 Java 版本的环境搭建）等。

TensorFlow 神经网络基础和各种模型部分（第 4 章到第 6 章）

从基础的神经元开始，逐步讲解单层神经网络、多层神经网络、卷积神经网络、循环神经网络等，在每个神经网络中还讲解其附带的各种数据集、模型和变体，比如 BP 神经网络、AlexNet 模型、LSTM 等。通过这些内容，读者可以对神经网络等知识



有初步认识，并能把握其整体脉络。

TensorFlow 的常用技术（第 7 章到第 9 章）

主要讲解 TensorFlow 对于数据集的制作和处理，以及在 TensorBoard 的模式下可视觉观察模型的各种参数变化，并讲解了支持向量机的概念。之所以把支持向量机放在这个部分，是因为支持向量机目前在深度学习领域是非常常用的，比如目标检测和分类任务。通过学习这一部分的内容，读者可以对 TensorFlow 在实际应用过程中的常用技术有一定的了解。

TensorFlow 的实际应用（第 10 章到第 14 章）

主要讲解如何使用 TensorFlow 在实际生活中解决问题，并讲解 TensorFlow 打包发布的各种方法，以及在 GPU 环境下如何使用 TensorFlow。另外，在这一部分，我们也讲到目标检测，读者可以将其思路用于实际工业生产领域当中。

初学 TensorFlow 的读者阅读本书时需要具有一定的 Python 基础，并建议从第 1 章开始，如果您有 GPU 版本的服务器或主机，则可以选择先看第 13 章，而不必阅读第 2 章；另外本书中含有一部分公式的推导，例如第 6 章循环神经网络会讲解关于 BP 神经网络的推导过程，这需要您有一定的数学基础，但是如果您看不懂数学推导也不会耽误对后面章节的理解和学习；如果您从事图像处理相关的工作和研究，可以着重看第 5 章卷积神经网络和第 8 章 TensorFlow 中的数据操作，通过这两章内容，我们可以建立自己的数据集，并将数据集应用到自己的模型中。如果您想了解各个公开数据集的相关信息，以及如何使用公开的数据模型，可以参考第 5 章卷积神经网络。

针对已经从事 TensorFlow 相关工作的读者，本书中含有大量的实例，以及作者在实际开发过程中的心得体会，并且循序渐进地讲解了公式推导部分，在讲解知识点的时候会推荐知识点，例如讲解 RNN，同时会讲解为什么会使用 RNN，RNN 是由什么样的神经网络算法（BP 神经网络）演变而来的，又在其中加入了什么样的内容（时序处理），以及这些算法的推导原理等。通过对一个网络的深入研究，能够使您在使用相关模型时了解其本质。



致谢

本书的顺利出版离不开大家对我的帮助。

在这里，首先我要感谢电子工业出版社博文视点的孙学瑛老师对我的信任及支持，正是由于孙老师的支持，才能有此书的顺利出版。其次我要感谢我的家人在我写书的这近一年时间内给我的支持和理解，尤其是我的妻子，在怀有身孕的情况下还帮助我分担了一些家庭中的琐事，使得我的写作能够顺利完成。

另外，还要感谢华中科技大学的胡中旭博士对本书关于神经网络及强化学习部分给出的建议和支持；感谢南京大学黄继鹏对于 RNN 部分写诗机器人的支持；感谢格力电器大数据中心的刘艳国帮助我实现了 Android 相关章节代码的调试工作；感谢韦家弘同学对于 RNN 相关内容的帮助和支持。

除此之外，本书还参考了网上的一些图例和部分代码，由于大多无法找到其出处，所以无法一一列举，在此感谢相关作者。

最后还要感谢所有在本书编写过程中给予我支持和帮助的人们，正是你们的帮助才使本书得以面世。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书如提供示例代码及资源文件，均可在 [下载资源](#) 处下载。
- **提交勘误**：您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/34565>



目 录

第 1 章 人工智能与深度学习	1
1.1 人工智能与机器学习	1
1.2 无处不在的深度学习	6
1.3 如何入门深度学习	7
1.4 主流深度学习框架介绍	13
第 2 章 搭建 TensorFlow 环境	15
2.1 基于 pip 安装	15
2.1.1 基于 Windows 环境安装 TensorFlow	15
2.1.2 基于 Linux 环境安装 TensorFlow	22
2.2 基于 Java 安装 TensorFlow	24
2.3 安装 TensorFlow 的常用依赖模块	27
2.4 Hello TensorFlow	30
2.4.1 MNIST 数据集	30
2.4.2 编写训练程序	32
2.5 小结	35
第 3 章 TensorFlow 基础	36
3.1 TensorFlow 的系统架构	36
3.1.1 Client	37
3.1.2 Distributed Master	38
3.1.3 Worker Service	39



3.1.4	Kernel Implements	39
3.2	TensorFlow 的数据结构——张量	39
3.2.1	什么是张量	39
3.2.2	张量的阶	40
3.2.3	张量的形状	40
3.2.4	数据类型	41
3.3	TensorFlow 的计算模型——图	42
3.3.1	计算图基础	42
3.3.2	计算图的组成	43
3.3.3	计算图的使用	45
3.3.4	小结	48
3.4	TensorFlow 中的会话——Session	48
第 4 章	TensorFlow 中常用的激活函数与神经网络	50
4.1	激活函数的概念	50
4.2	常用的激活函数	51
4.2.1	Sigmoid 函数	51
4.2.2	Tanh 函数	53
4.2.3	ReLU 函数	55
4.2.4	Softplus 函数	57
4.2.5	Softmax 函数	58
4.2.6	小结	59
4.3	损失函数的概念	60
4.4	损失函数的分类	63
4.5	常用的损失函数	65
4.5.1	0-1 损失函数	65
4.5.2	Log 损失函数	66
4.5.3	Hinge 损失函数	69
4.5.4	指数损失	70
4.5.5	感知机损失	70
4.5.6	平方（均方）损失函数	71



4.5.7	绝对值损失函数	71
4.5.8	自定义损失函数	71
4.6	正则项	72
4.6.1	L0 范数和 L1 范数	72
4.6.2	L2 范数	73
4.6.3	核范数	74
4.7	规则化参数	76
4.8	易混淆的概念	76
4.9	神经网络的优化方法	77
4.9.1	梯度下降算法	77
4.9.2	随机梯度下降算法	79
4.9.3	其他的优化算法	80
4.9.4	小结	84
4.10	生成式对抗网络 (GAN)	84
4.10.1	CGAN	96
4.10.2	DCGAN	97
4.10.3	WGAN	98
4.10.4	LSGAN	99
4.10.5	BEGAN	100
第 5 章	卷积神经网络	102
5.1	神经网络简介	102
5.1.1	神经元与神经网络	102
5.1.2	感知器 (单层神经网络) 与多层感知器	104
5.2	图像识别问题	108
5.3	常用的图像库介绍	111
5.4	卷积神经网络简介	114
5.4.1	CNN 的基本原理与卷积核	115
5.4.2	池化	116
5.4.3	再探 ReLU	118
5.5	CNN 模型	119

5.5.1 LeNet-5 模型	119
5.5.2 AlexNet 模型	123
5.5.3 Inception 模型	130
5.6 用 CNN 实现 MNIST 训练	147
第 6 章 循环神经网络	151
6.1 初识循环神经网络	151
6.1.1 前馈神经网络	152
6.1.2 神经网络中的时序信息	158
6.2 详解循环神经网络	159
6.3 RNN 的变种——双向 RNN	162
6.4 One-Hot Encoding	165
6.5 词向量和 word2vec	166
6.5.1 CBOW 模型	167
6.5.2 Skip-Gram 模型	168
6.6 梯度消失问题和梯度爆炸问题	169
6.6.1 梯度下降	170
6.6.2 解决梯度消失和梯度爆炸问题的方法	172
6.7 RNN 的变种——LSTM	179
6.8 写诗机器人	189
第 7 章 TensorFlow 的可视化	196
7.1 TensorBoard 简介	196
7.2 生成和使用 TensorBoard	200
7.3 TensorBoard 的面板展示	208
7.4 小结	223
第 8 章 TensorFlow 中的数据操作	224
8.1 制作 TFRecords 数据集	224
8.2 Dataset API 介绍	230

8.3 TensorFlow 中的队列	233
第 9 章 支持向量机 (SVM)	240
9.1 什么是支持向量机	240
9.2 计算最优超平面	242
9.3 TensorFlow 实现线性 SVM	243
9.4 非线性 SVM 介绍	247
9.5 使用 TensorFlow 实现非线性 SVM 分类器	250
第 10 章 TensorFlow 结合 Flask 发布 MNIST 模型	258
10.1 Flask 框架介绍	258
10.2 训练 MNIST 模型	259
10.3 小结	275
第 11 章 TensorFlow 模型的发布与部署	276
11.1 TensorFlow Serving 的前导知识	276
11.2 TensorFlow Serving 模型打包	280
11.3 TensorFlow Serving 模型的部署和调用	284
第 12 章 TensorFlow Lite 牛刀小试	285
12.1 什么是 TensorFlow Lite	285
12.2 如何使用 TensorFlow Lite 模型	287
12.3 TensorFlow Lite 与 Android 结合实现图像识别	290
第 13 章 TensorFlow GPU	296
13.1 什么是 GPU	296
13.2 GPU 的选择	297
13.3 搭建 TensorFlow GPU	299
13.3.1 在 Windows 上搭建 TensorFlow GPU	299
13.3.2 在 Linux 上搭建 TensorFlow GPU	307

13.4 使用 TensorFlow GPU 进行训练	311
第 14 章 TensorFlow 与目标检测.....	317
14.1 传统目标检测方法	317
14.2 RCNN 介绍	319
14.3 Fast-RCNN	321
14.4 Faster-RCNN	325
14.5 YOLO	328
附录 A TensorFlow 历代版本更新内容	354
A.1 TensorFlow 1.3 版本更新内容	354
A.2 TensorFlow 1.4 版本更新内容	355
A.3 TensorFlow 1.5 版本更新内容	356
A.4 TensorFlow 1.6 版本更新内容	356
A.5 TensorFlow 1.7 版本更新内容	357
A.6 TensorFlow 1.8 版本更新内容	357
A.7 TensorFlow 1.9 版本更新内容	358

第 1 章

人工智能与深度学习

1.1 人工智能与机器学习

早在远古时期，人们就希望通过机器来减轻劳动量，凭借智慧制造出了各种各样的工具来代替劳作。后来人们把这些能够利用所产生的能量达到特定目的的工具、装置或者设备统称为机器。制造机器的最初目的仅仅是为了减轻人类繁重的劳动，而不是代替人类，这是因为大部分机器必须通过人类的操作才能完成任务。从传统的简单机械，例如爬犁、锄头到后来的工业化机器，例如蒸汽机、流水线，都要依赖人类的操作才能运转，也就是说这些机器是在人类的控制下并赋予特殊的指令才能完成各种各样的操作的。如果给机器赋予能够自主思考的能力，给定一个特定的目标后，机器通过自主学习或通过观察周围的环境来做出自我判断，那么机器就会变得更加“聪明”。也就是说，我们可以通过技术使机器能够接近或达到人类的智慧，这样的机器才能更受人们的喜爱，我们把这样的一种智慧或者智能称为人工智能（**Artificial Intelligence, AI**），或称机器智能。

人工智能从字面意义上来讲可以分成两个含义，即“人工”和“智能”。“人工”一词非常好理解，即通过人力来完成我们所要达到的目的。但对于“智能”一词，我们可以将其翻译或理解成多种意思：对于人类来讲，人体本身就是一个智能体，因为人类具有很强的自主意识和思维能力；由人类所创造出来的含有高科技的事物也被称

为智能；甚至一些无意识的行为或精神也是智能的一种体现。因此，我们很难去通过一种很强的界限来区别出什么是智能，什么是“人工”所制造出来的“智能”了。人工智能的研究领域多种多样，这不仅仅体现在通过人类的智慧所制造出来的具有自主意识的事物，同时还可以包括人类思维和意识的本身，但就目前来讲，我们所理解的人工智能通常是指通过计算机的程序来实现人类智能的一种技术手段，本书所指的人工智能也是如此。

1946 年是计算机发展史上最重要的一年。1946 年 2 月，由美国宾夕法尼亚大学所研发的第一台现代电子计算机埃尼阿克（ENIAC）诞生，它是图灵完全的电子计算机，能够通过编程解决各类计算机问题。ENIAC 由 18 000 个左右的电子管、1 500 个继电器和 10 000 个电容器组成，重量达 27 吨，占地面积约 170 平方米。ENIAC 的诞生具有重大的意义，它对人工智能领域的发展起到了深远且积极的影响。图 1-1 所示为世界上第一台现代电子计算机 ENIAC。

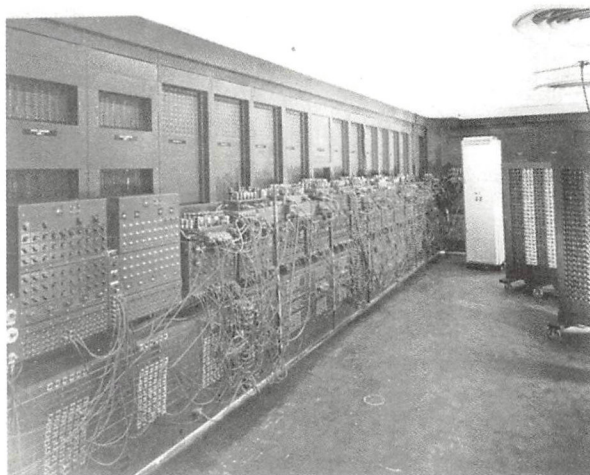


图 1-1

如果说 1946 年是计算机发展史上最重要的一年，那么 1956 年就是人工智能发展史上最重要的一年。在 1956 年的达特茅斯会议上，来自麻省理工学院的约翰·麦卡锡提出：“人工智能就是要让机器的行为看起来就像人所表现出的智能行为一样。”也就是从那时起，“人工智能”一词正式诞生，同时在会议上人工智能也被正式确立为

一门科学。

从广义上划分，人工智能包含强人工智能和弱人工智能两个部分。强人工智能也叫作通用人工智能（Artificial General Intelligence），实质具备与人类同等智慧、或超越人类的人工智能，能够表现出人类所具有的全部智能行为。强人工智能也是目前人工智能研究的主要目标之一，自动规划、使用自然语言进行沟通、自动推理等都属于强人工智能研究领域范围。而弱人工智能是指不能进行真正的推理和解决问题的智能机器，这些机器看起来像是智能的，但其实并没有拥有真正的智能，也不会有自主意识。

人工智能有非常广泛的研究分类，例如机器人学、情感计算、机器学习等，其中机器学习是人工智能领域中一个非常重要的分支。

机器学习（Machine Learning, ML）的主要目的就是为了让机器从用户输入的数据等获取知识，从而使机器能够自动地判断和输出相应的结果。机器学习的方法有很多种，主要分为监督学习和无监督学习。其中监督学习是指利用一组已知特征的数据进行训练，使机器能够根据已知的特征和规律进行相应的总结，从而自主建立相应的模型，并使用这个模型根据相应的输入值进行预测，得出最终所需要输出的结果；而无监督学习（也称非监督式学习）是另一种机器学习的形式，其特点就是能够使机器在没有特定的标注的数据集的基础上进行训练，让机器自己寻找规律，建立相应的模型，从而使这个模型具备预测的能力，根据输入得到相应的输出。这里的机器一般指的是计算机。

一般来讲，在监督学习中，训练集要求包含明确的输入和输出，这里的输入和输出一般是指特征和目标。例如我们想训练机器识别一条狗，那么需要在给定的数据集中明确地指出在这张图片中哪个部分是一条狗，如图 1-2 所示。而非监督学习中，则不需要在训练集中明确地表明输入和输出，也不需要告诉机器要训练的部分位于数据中的哪个位置，只需要将包含图片的训练数据集给到机器，机器就能自动地从这些数据中学习到相应的特征，并将这些特征加以整合，等到一个新的数据被传入之后，就能够根据模型进行相应的预测。

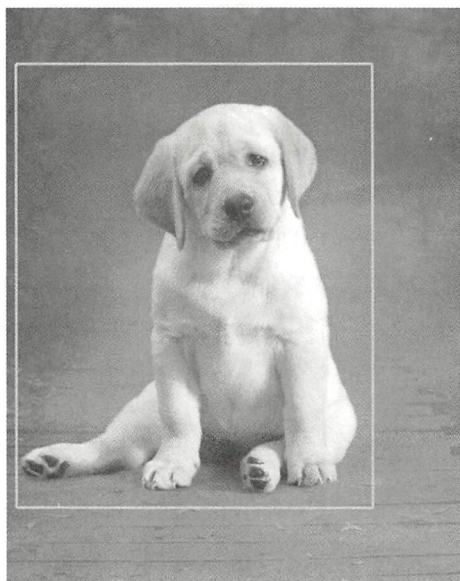


图 1-2

其实除监督学习和非监督学习外，还有两种很常见的学习方式：半监督学习和强化学习。所谓半监督学习，顾名思义，就是介于监督学习和无监督学习两者之间的学习方式，同时使用大量的未标记的数据和大量的已标记的数据进行数据模型的建立，从而达到预测的效果。当使用半监督学习时，参与的人员可尽量少，而且还能带来比较高的准确性，因此，半监督学习目前正越来越受到人们的重视。而强化学习则强调如何基于环境行动，以取得最大化的预期利益。强化学习和标准的监督学习之间的区别在于，它并不需要出现正确的输入/输出对，也不需要精确校正次优化的行为。强化学习更加专注于在线规划，需要在探索（未知领域）和遵从（现有知识）之间找到平衡。

机器学习不是一门单纯的学科，实际上，我们可以将机器学习理解为很多门学科的综合和交叉，涉及概率论、线性代数、统计学、算法复杂度等多门理论学科。机器学习的核心是如何使用计算机模拟和实现人类的学习行为，以获得新的知识和技能，并将已有的知识进行重新组织和梳理，使其结构不断改善和更新，从而提高自身的性能和价值。

从上个世纪末开始，机器学习已经在世界上崭露头角。早在 1997 年 5 月，IBM 的“深蓝”超级计算机就已经可以击败当时的国际象棋世界冠军卡斯巴罗夫。而在接下来的这 20 年左右的时间里，机器学习又在图像识别、自然语言处理等多个领域取得了非常惊人的进展。例如在语音识别领域，将语音识别成文字；在图像处理领域，进行人脸识别等操作。可以说，目前，机器学习已经充斥着整个世界。

在大部分的人工智能任务中，我们都需要对特征进行处理，但是往往由于环境等因素，导致有些特征无法进行直接处理或者很好处理。为了解决这个问题，在机器学习中又引入了另外一个概念——**表征学习 (Representation Learning)**。表征学习又叫特征学习或表示学习，顾名思义，表征学习的意义就在于它不仅能够使计算机学习和使用特征，而且还学习到了如何提取特征，并自己告诉自己该如何学习。在机器学习中，表征学习是学习一个特征的技术的集合，在机器学习东西的任务中，对输入的数据一般都会有一定的要求，其最大的要求就是便于机器计算和处理。但是在现实世界中，很多数据并不能满足这一要求，例如在进行视频或者音频对象的输入时，其数据的不确定性就会变得非常大，也比较复杂，因而造成处理的难度相对较大，且冗余、多变。表征学习就能够很好地解决此类问题。

表征学习同样也可以被分为监督学习和无监督学习两类，这和机器学习非常类似。在表征学习的监督特征学习中，被标记过的数据会被当作特征用来学习，例如神经网络、多层感知器等；而在无监督特征学习中，未被标记过的数据会作为一个整体被当作一个特征用来学习，例如无监督的字典学习、自编码器等都是其典型的代表。

表征学习的目标是寻求更好的表示方法并创建更好的模型来从大规模未标记数据中学习这些表示方法。表示方法来自神经科学，并松散地创建在类似神经系统中的信息处理和对通信模式的理解上，如神经编码，试图定义拉动神经元的反应之间的关系，以及大脑中的神经元的电活动之间的关系。

我们暂且可以粗略地认为，表征学习可以表示任何的特征，甚至可以夸张地认为表征学习可以表示一切。然而这种愿景是美好的，事实却并非如此，对于固定的形态，我们确实可以使用表征学习的方式进行表示，但是如果要表示一台汽车的车轮，不同车的车轮不同，即使同一台车在不同的光照角度下的阴影也有很大的差别，这个时候，

我们就无法使用定性定量的方式将车轮表示出来，用表征学习进行特征的表示可能就会变得非常困难。针对同一个物体在不同环境下所存在的不同状态，我们将其认定为是由数据的变差因素（**Factors of Variation**）所导致的。

1.2 无处不在的深度学习

为了解决表征学习所面临的问题和困难，研究人员提出了一个新的概念——深度学习（**Deep Learning**）。

深度学习是机器学习的一个分支，也是目前人工智能领域和机器学习领域中最重要的一個分支，它是一种试图使用包含复杂结构或由多重非线性变换构成的多个处理层对数据进行高层抽象的算法。同时，深度学习也是机器学习中一种基于对数据进行表征学习的算法。深度学习可以使用其他较简单的表示来表达复杂的表示，这句话听起来非常拗口，但是实际上，我们使用深度学习的主要目的就是使用表征学习将表示起来困难的问题简单化。

深度学习最早的雏形要追溯到久远的人工神经网络，而实际上，深度学习这个概念是 2006 年左右由 Geoffrey Everest Hinton 和 Ruslan Salakhutdinov 提出的。当时他们提出了一种在前馈神经网络中进行有效训练的算法，这一算法将网络中的每一层视为无监督的受限玻尔兹曼机，再使用有监督的反向传播算法进行调优。

深度学习自出现以来就已经在很多领域进行了广泛的应用，尤其是在计算机视觉、文本处理和语音识别领域中发挥着非常积极的作用，成为各种领先系统中的一部分。例如在图像识别中，利用深度学习技术可以快速地进行目标检测和识别，并且相对于传统方法来说，其精度和速度有了大幅度的提高；到目前为止，已经有数十种基于深度学习而形成的神经网络，例如卷积神经网络、循环神经网络、深度置信网络、递归神经网络等，已经被应用在了各种各样的场景下。

深度学习实际上是机器学习中一个比较重要的分支，因此，一个深度学习网络的训练方法也会被分为有监督训练和无监督训练，其训练的方式和思路与机器学习是类似的。相对传统的机器学习而言，使用深度学习进行模型的建立和预测所得到的准确

率和效率都会变得更高。

无论深度学习还是机器学习，或者是传统的人工智能，到目前为止，最大的阻碍实际上还是在计算机硬件上，在传统的计算机运算中使用的是 CPU，而在机器学习或者深度学习领域中，除了使用 CPU 进行数据集的训练之外，我们更加推荐使用 GPU 进行训练。关于 GPU 的相关内容我们会在后续的章节中进行详细的讲解。

1.3 如何入门深度学习

深度学习的框架有很多种，无论是使用哪种深度学习框架，都离不开了解框架本身的特性和接口，以及具有一定的算法和神经网络相关知识。因此，要想学好深度学习框架，需要经过以下几个步骤。

了解深度学习相关概念，知道深度学习的具体作用

要想成为一名合格的深度学习开发者，首先要清楚：深度学习到底是什么？深度学习具体有什么用？使用深度学习能够做些什么事情？

深度学习的概念起源于人工神经网络，通过组合低层的特征形成更加抽象的高层表示，以此来发现数据的特征分布。我们可以把深度学习的过程看作人脑工作和学习的一个过程。

举个例子：

当我们拿到一个苹果的时候，一眼就能够将它辨识出来，但是，我们为什么会认为它一定是个苹果，而不是个香蕉或者其他水果呢？首先当眼睛看到的图片传入大脑时，我们首先会看到，这个物体是圆圆的、红红的，上下两侧是向里面凹陷进去的，顶部有一个大约 1 厘米的柄；然后再从中间将其切开，发现里面有果核；接着尝一口，发现味道是酸甜酸甜的，于是我们就认为这是一个苹果，而不是香蕉或者鸭梨等其他水果。如图 1-3 所示。

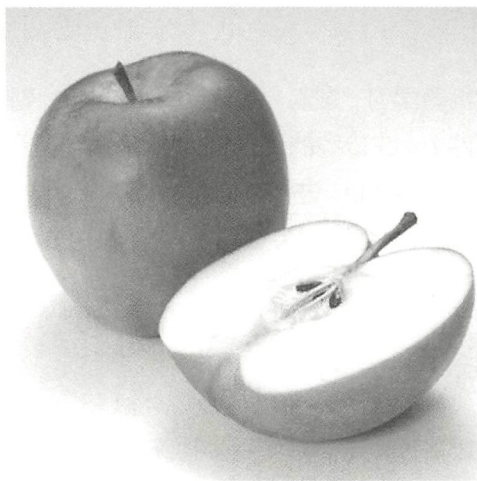


图 1-3

刚才我们之所以能够很容易地判断出这个苹果，是因为知道了苹果的一些相关特征，单独符合这些特征的物体有很多，但是我们把这些特征组合起来，就是苹果的独有特征，因此，我们就可以根据这些特征判断出：这是苹果。

同样地，深度学习框架也是如此，我们在前期只需要把每种事物一系列特征给到机器，机器就能根据我们所给的一系列特征进行加工和判断，以及在各种事物之间、各种特征之间产生联系。当输入的数据越多，这些特征之间的联系就越为紧密，机器对事物的判断能力就会越强，于是我们就说机器的学习能力越强，越来越接近人类的思维，这也正是我们使用深度学习来进行人工智能研究的本质所在。

在当今各个领域内，我们都有相对成熟的深度学习相关案例，例如：目前很多国家都已经实现了自动驾驶技术，只需要告诉汽车你要去的目的地，汽车就可以根据用户设置的目的地，自动驾驶，在行驶过程中遇到行人、红绿灯，以及其他车辆等，都会根据其车速和距离自动启动制动功能，同时也会根据地图系统进行精确的方向控制；在医疗领域，也有相关的医疗人员利用深度学习技术，让计算机帮助人类判断患者的医学影像，并告知其影像是否可能或已经发生了病变，以及针对病变的具体解决办法，以辅助医生进行治疗；在银行、金融等行业，也在使用深度学习技术进行欺诈检测等。

具有一定的数学功底

因为在深度学习和机器学习的领域中，我们会用到大量的数学知识，例如：利用矩阵乘法对类型进行抽象运算；在利用 GPU 运算的时候，图像可以表示为像素组，GPU 再来处理并行的整个像素矩阵；在深度学习的过程中，经常使用的一个算法叫作梯度下降算法，而梯度下降算法主要用到了高等数学当中的导数求导的相关知识；在进行深度学习的过程中，我们经常会做一些预测，例如预测未来三天的天气情况，未来几个小时内的股票走势情况，此时都会用到概率论中的累积分布知识、随机概率事件、概率密度估计函数等。

所以要想在深度学习方面有较深的突破，那么就必须对深度学习中所涉及的数学知识有一定的了解。面对不同的情况，我们需要使用什么样的数学模型进行深度学习，对于学好深度学习来讲是非常关键的。

了解基本的深度学习理论

要想学好深度学习框架，就必须对深度学习的相关理论和所用到的神经网络有一定的认识 and 了解，例如：

- 利用长短期记忆网络（Long Short-Term Memory, LSTM）处理和预测时间序列中间隔和延迟非常长的重要事件；
- 利用卷积神经网络（Convolutional Neural Network, CNN）进行图像处理；
- 利用回声状态网络（Echo State Network, ESN）进行机器人控制，运动目标检测等。

另外，在深度学习相关算法和模型方面，也需要有一定的认识，例如：

- 利用 KMeans 算法进行基于距离划分的聚类操作；
- 利用 SVD（Singular Value Decomposition, SVD）算法进行矩阵奇异值的分解；
- 利用朴素贝叶斯算法进行中文文本的分类等。

只有对深度学习的基本理论和算法有所了解，才能更加得心应手地利用现有的模型做深度学习的研究，并且自己开发新的深度学习模型。

掌握一定的计算机理论和编程工具

要想深入地研究深度学习领域的相关知识，训练出一个可靠的模型，大量的数据支撑是必不可少的。大数据时代背景下，深度学习作为依靠数据支撑的主要技术，与 Hadoop, Spark 等数据处理框架的结合是不可避免的，因此，还需要对目前的大数据处理技术，以及计算机的相关理论知识做一定的积累。

计算机理论中关于分布式部署与分布式计算的知识，可以帮助我们更好地提炼出深度学习中所需要的数据，以及在集群环境下部署深度学习框架。

我们还要了解一些科学计算相关的语言和工具，比较常见的科学计算相关语言有 Python, Erlang, Haskell 等，比较常见的科学计算工具有 NumPy, MATLAB 等。

了解框架的开发语言

要想对深度学习有深入的研究，不仅要去研究深度学习框架的本身，还要了解深度学习框架所应用的开发语言，例如本书我们所使用的深度学习框架 TensorFlow 支持 Python, C++, Java 三种编程语言。本书所使用的 Python 语言，也是 TensorFlow 开发的主流编程语言。

Python 语言是一种面向对象的解释型计算机程序设计语言，具有丰富和强大的库，能够把用其他语言制作的各种模块很轻松地连接在一起。我们可以使用 NumPy, Matplotlib, Pandas 等第三方库，很轻松地实现对线性代数和矩阵的操作，为我们使用 TensorFlow 框架开发提供了极大的便利。

只有熟练使用 Python 语言及其第三方库，才能够开发出更加强大的深度学习产品。

给自己定一个可以达到的小目标

进行深度学习框架研究的人一般来讲主要分为两大类：第一类是想要通过深度学习框架进行深度学习相关应用开发的工程师；第二类是想在深度学习领域有相对较深的突破，并能够对深度学习框架底层有一定研究，甚至在此基础上进行源码的修改、封装和算法的改进。

针对第一类,建议主要从深度学习框架本身入手(例如本书的 TensorFlow 框架),了解框架的基本特点,研究框架的 API 使用,以及通过书本或者官方文档的代码例子快速上手。面对一些复杂的算法及其模型,只要知道这个模型是做什么用的,怎么去用就可以了,没有必要针对模型中的每个算法和模型进行非常深入细致的研究。此类读者,能够快速做出小的 Demo,提高自身的自信心才是关键所在。

针对第二类,建议在深入研究深度学习之前,首先要对线性代数、高等数学、概率论等数学知识有着相对全面并且深入的了解,其次要对所研究的方向(例如:文本处理、图像处理、分类、预测等)所涉及的技术点和相关方法有所了解,在研究深度学习与其结合的时候,要着重了解其工作原理,以及如何更好地优化相关的算法,使其运行效率达到一个相对理想的状态;要想细致深入地研究深度学习,精读国外著作和论文是必不可少的,要多去研究国外技术专家是如何进行研究的,以及在 GitHub 找到 TensorFlow 相关源码研习。

无论您是第一类读者还是第二类读者,在确立了自己的方向之后,要给自己定一个可以达到的小目标,例如:训练一个属于自己的聊天机器人,给老旧的黑白照片上色,引用或开发更加高级的深度学习算法模型,使其工作效率更高。只有这样,我们才能不断地进步。

研究前人总结的成果,不懂就要问

牛顿曾经说过:“如果说我看得比别人更近些,那是因为我站在巨人的肩膀上。”深度学习也是如此,如果想在深度学习领域有更高的成就,那么就一定要去学习业界精英们所留下的成果。一般来讲,我们在深入学习 TensorFlow 的时候,可以选择以下几种途径:

- 去 GitHub 上面下载业界精英们上传的源代码,首先要根据代码提供者们提供的文档将项目在本机运行起来,然后再根据自己感兴趣的点精读相关源码部分;
- 去 TensorFlow 的官方网站查看官方文档,因为,官方的文档永远是最新的,不过前提是要有良好的英文功底;
- 可以去阅读一些网上的博文、笔记及相关的论文文献,从中获得启发;
- 加入一些与 TensorFlow 相关的 QQ 技术群,关注群里面所讨论的技术问题,并经

常在上面进行交流，因为，你不懂的问题也许别人会懂，提出来可以互相探讨；别人提出的问题也许现在你还没有遇到过，但是总有一天，你也会遇到同样的问题。

多动手，多训练

作为一名技术人员来讲，有着扎实的基本功和丰富的理论固然重要，但是最重要的还是通过敲代码的方式将自己的思路转换成可运行的计算机程序。

在初期的时候，可以参照 TensorFlow 官方文档和本书所提供的相关文档，自己在本机将程序跑起来，然后试着去理解程序里面的每一行代码。在理解代码的基础上，尝试对代码进行小范围的更改，比如更改一个参数，或者更改一种公式，再跑一遍，看看所得到的结果与自己心中所设想的结果有没有差别，以及差距有多大，并根据计算机所运行的结果去思考，为什么计算机程序得出的结果是这个样子的。

当对 TensorFlow 框架及其 API 有了一定的了解之后，可以试着去 GitHub 下载一些其他人所开源的代码，然后依照上面的步骤，得到自己想要的结果。

当你认为自己看懂网上的代码已经没有什么问题了，可以试着自己写一些小的程序，比如训练自己的图像数据集分类。

查漏补缺，持续进步

在利用深度学习框架进行训练的过程中，你可能会遇到各种各样的问题和技术瓶颈，即使当你训练好了一个模型之后，在下次使用模型进行训练操作时，还会发现这个模型似乎又不满足当前的需求，需要不断地进行优化。这个时候，首先就要想到如何查漏补缺，看看自己模型中存在哪些问题，为什么上次没有想到，以及如何进行模型的调整，更改模型的参数，使自己训练的模型更贴近业务本身的需求。

1.4 主流深度学习框架介绍

Caffe

Caffe(Convolutional Architecture for Fast Feature Embedding)是一个清晰、高效、可读性高的深度学习框架，也是深度学习发展初期主流的工业级工具，在卷积神经网络(CNN)的实现和计算机视觉领域方面具有非常出色的表现，可以在多种设备上进行编译，具有非常好的跨平台特性，部署简单、快速；但是由于 Caffe 框架在最初设计的时候遗留了许多架构方面的问题，使得它对递归网络以及语言建模的支持并不是很好。正是由于架构设计的问题，Caffe 框架的图层必须使用 C++定义，模型必须使用 protobuf 定义，大大降低了框架使用的灵活性。除非现有的 Caffe 模型能够非常符合使用者的需要，否则目前在深度学习领域开发和研究人员很少会将其作为首选开发框架。

Theano

Theano 框架是一个完全使用 Python 语言编写的深度学习框架，并支持大多数的神经网络，对于循环神经网络(RNN)具有非常好的支持，可以很好地跨平台部署到 Windows 环境当中。但是由于此框架是使用 Python 语言编写的，而 Python 是一种解释型的编程语言，对于相对较大的模型开销非常大，运行效率相对较慢，使得 Theano 对于追求效率及相对较大的模型似乎显得望尘莫及。

Torch

Torch 框架是一个由 Facebook 主推的深度学习框架，运行效率较高，对于长短期记忆网络(LSTM)具有非常好的支持，但是由于此框架的开发语言为 Lua 语言，而在深度学习领域，Lua 语言相对小众化，所以 Torch 框架一直无法得到较好的发展。

TensorFlow

TensorFlow 框架是 Google 公司于 2015 年 11 月开源的深度学习框架，Tensor(张量)意味着 N 维数组，Flow(流)意味着基于数据流图的计算，TensorFlow 即为张量从流图的一端流动到另一端的计算过程。TensorFlow 框架结构清晰，开发者能够快速

上手使用。TensorFlow 支持大部分先进的神经网络，如卷积神经网络（CNN）、循环神经网络（RNN）、长短期记忆网络（LSTM）等。TensorFlow 使用了向量运算的符号图方法，使得新网络的指定变得相当容易。程序开发人员可以使用 C++、Python、Java 作为 TensorFlow 框架的编程语言，利用官方提供的 API 接口，快速地开发自己的深度学习模型。图 1-4 所示为 TensorFlow 框架的 LOGO。



图 1-4

第 2 章

搭建 TensorFlow 环境

前面介绍了深度学习与人工智能的基础知识，以及 TensorFlow 框架的相关特性。本章将带领大家一步一步地将 TensorFlow 环境搭建起来。

2.1 基于 pip 安装

2.1.1 基于 Windows 环境安装 TensorFlow

TensorFlow 从 0.12.0 版本开始支持在 Windows 操作系统下安装，在 Windows 操作系统下安装 TensorFlow 的必备环境有：

1. Windows 7 版本以上的 64 位操作系统；
2. 64 位 Python 3.6.X 的发行版本；
3. TensorFlow 0.12.0 以上的版本；
4. GPU 版本需要 CUDA8 和 CUDNN5.1 以上版本。



第一步 安装 Python

TensorFlow 在 Windows 上只支持 64 位 3.5.X 以上版本的 Python。在选择 Python 发行版的时候，可以选择 Python Releases for Windows 版本或者 Anaconda for Python 版本。由于 Anaconda for Python 版本中包含了深度学习所需要的绝大多数的第三方库，所以推荐使用 Anaconda for Python 作为 Windows 下的开发环境。下面开始一步一步地安装 Anaconda。

(1) 下载 Anaconda

Anaconda 的下载地址为 <https://www.anaconda.com/download/>，打开浏览器，输入网址后进入 Anaconda 的下载界面，如图 2-1 所示。

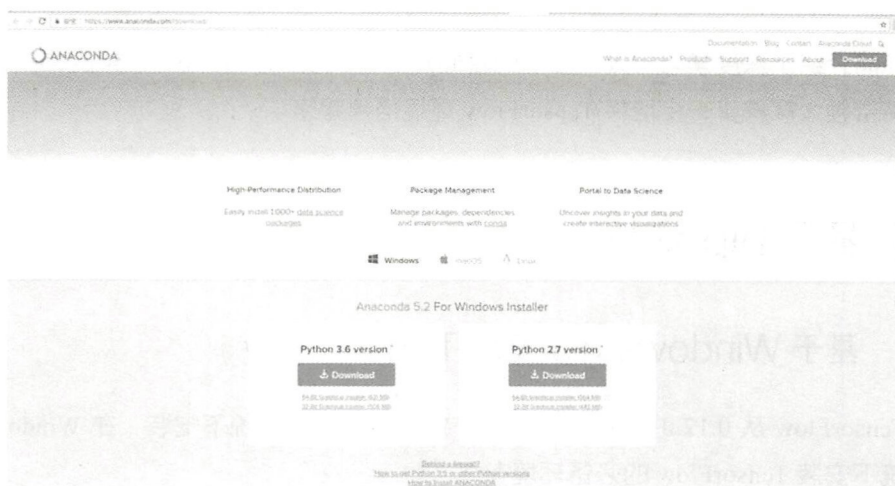


图 2-1

由于 TensorFlow 只支持在 64 位 Python 下进行开发，所以一定要选择 64-bit 版本，千万不要选择 32-bit 版本，如图 2-2 所示。





图 2-2

即进入下载列表页面，选择 Anaconda3-5.2.0-Windows-x86_64.zip，等待下载完成即可，如果由于网络原因导致下载多次失败，可以尝试使用清华大学的源地址进行下载，如图 2-3 所示。

← → 安全 https://mirrors.tuna.tsinghua.edu.cn/anaconda/archive/		
Anaconda3-5.0.0-Linux-x86_64.sh	534.0 MiB	2017-10-03 00:34
Anaconda3-5.0.1-Linux-x86.sh	431.0 MiB	2017-10-26 00:41
Anaconda3-5.0.1-Linux-x86_64.sh	525.3 MiB	2017-10-26 00:42
Anaconda3-5.0.1-MacOSX-x86_64.pkg	568.9 MiB	2017-10-26 00:42
Anaconda3-5.0.1-MacOSX-x86_64.sh	491.0 MiB	2017-10-26 00:42
Anaconda3-5.0.1-Windows-x86.exe	430.4 MiB	2017-10-26 00:44
Anaconda3-5.0.1-Windows-x86_64.exe	514.8 MiB	2017-10-26 00:45
Anaconda3-5.1.0-Linux-ppc64le.sh	285.7 MiB	2018-02-15 23:22
Anaconda3-5.1.0-Linux-x86.sh	449.7 MiB	2018-02-15 23:23
Anaconda3-5.1.0-Linux-x86_64.sh	551.2 MiB	2018-02-15 23:24
Anaconda3-5.1.0-MacOSX-x86_64.pkg	594.7 MiB	2018-02-15 23:24
Anaconda3-5.1.0-MacOSX-x86_64.sh	511.3 MiB	2018-02-15 23:24
Anaconda3-5.1.0-Windows-x86.exe	435.5 MiB	2018-02-15 23:26
Anaconda3-5.1.0-Windows-x86_64.exe	537.1 MiB	2018-02-15 23:27
Anaconda3-5.2.0-Linux-ppc64le.sh	288.3 MiB	2018-05-31 02:37
Anaconda3-5.2.0-Linux-x86.sh	507.3 MiB	2018-05-31 02:37
Anaconda3-5.2.0-Linux-x86_64.sh	621.6 MiB	2018-05-31 02:38
Anaconda3-5.2.0-MacOSX-x86_64.pkg	612.1 MiB	2018-05-31 02:38
Anaconda3-5.2.0-MacOSX-x86_64.sh	523.3 MiB	2018-05-31 02:39
Anaconda3-5.2.0-Windows-x86.exe	506.3 MiB	2018-05-31 02:41
Anaconda3-5.2.0-Windows-x86_64.exe	631.3 MiB	2018-05-31 02:41

图 2-3

(2) 安装 Anaconda

下载完成之后，双击下载后的安装文件，进行安装，如图 2-4 所示。

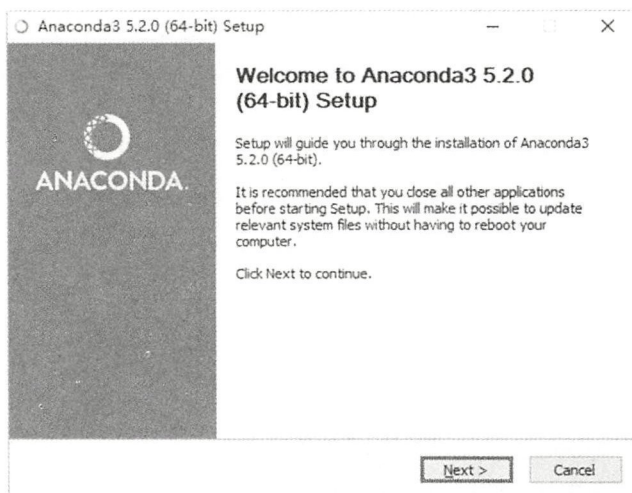


图 2-4

单击 Next>按钮，弹出协议界面，单击 I Agree，进入选择安装类型的界面，如图 2-5 所示。

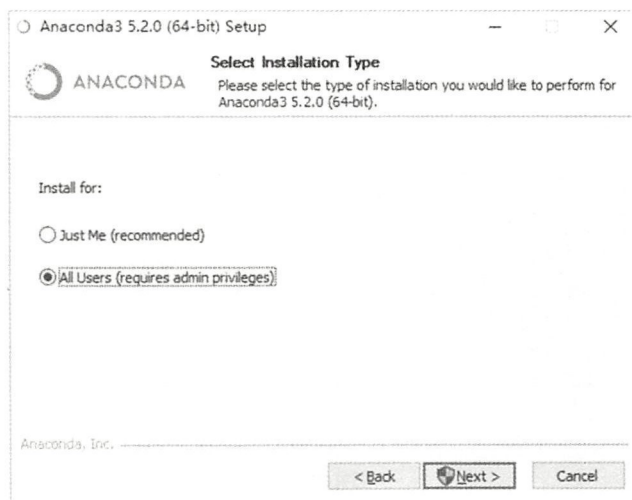


图 2-5

在选择安装类型的界面中，建议选择 All Users(requires admin privileges)，此项说明在使用计算机的所有用户都可以使用 Anaconda 开发环境，然后单击 Next>按钮进

入安装位置选择界面，选择路径后，单击 Next>按钮继续安装，如图 2-6 所示。

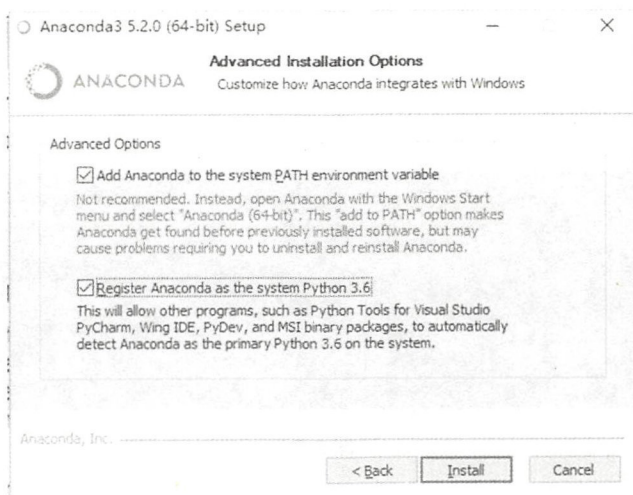


图 2-6

在安装选项中，第一个选项是告诉我们将 Anaconda 加入系统的环境变量里，第二个选项是告诉我们使用 Python 3.6 注册 Anaconda 到系统中，建议两个选项全部勾选，然后单击 Install 按钮进行安装，等待大约 5 分钟，整个程序安装完成。

接下来，进入控制台，输入 `conda --version`，如果出现如图 2-7 所示的界面，则说明 Anaconda 安装成功，接下来就可以进行 TensorFlow 的安装了。

```
C:\Users\Administrator>conda --version
conda 4.4.8
```

图 2-7

第二步 安装 TensorFlow

刚刚已经成功安装了 Anaconda 环境，接下来就要在 Anaconda 环境下安装 TensorFlow 了。

TensorFlow 无论在哪个环境下都分为 CPU 和 GPU 两个版本，其中 GPU 只支持 NVIDIA (英伟达) 的芯片，也就是俗称的“N 卡”，并且需要支持 CUDA 和 cuDNN；

TensorFlow 进阶指南

基础、算法与应用

CPU 版本并无其他限制，一般推荐使用 Intel 系列的 CPU。

在确保以上信息之后，下面将开始安装 TensorFlow。首先通过 `conda create -n tensorflow-t python=3.6` 命令在 Anaconda 下创建一个虚拟的 TensorFlow 环境，如果第一次使用，则会出现如图 2-8 所示的提示，输入 Y 继续安装：

```
The following NEW packages will be INSTALLED:
ca-certificates: 2018.4.16-0      https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
certifi: 2016.2.28-py36_0        https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
openssl: 1.0.2c-vc14_0           https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge [vc14]
pip: 9.0.1-py36_1               https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
python: 3.6.2-0                   https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
setuptools: 36.4.0-py36_1        https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
vc: 14-0                         https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
vs2015_runtime: 14.0.25420-0     https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
wheel: 0.29.0-py36_0            https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
wincertstore: 0.2-py36_0        https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free

Proceed ([y]/n)?
```

图 2-8

当出现如图 2-9 所示的界面，则表示虚拟环境创建成功。

```
The following NEW packages will be INSTALLED:
ca-certificates: 2018.4.16-0      https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
certifi: 2016.2.28-py36_0        https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
openssl: 1.0.2c-vc14_0           https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge [vc14]
pip: 9.0.1-py36_1               https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
python: 3.6.2-0                   https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
setuptools: 36.4.0-py36_1        https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
vc: 14-0                         https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
vs2015_runtime: 14.0.25420-0     https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
wheel: 0.29.0-py36_0            https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
winertstore: 0.2-py36_0          https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free

Proceed ([y]/n)? y

Preparing transaction: done
Verifying transaction: done
Executing transaction: done

#
# To activate this environment, use:
# > activate tensorflow-t
#
# To deactivate an active environment, use:
# > deactivate
#
# * for power-users using bash, you must source
#
```

图 2-9

输入 `activate tensorflow-t` 激活 TensorFlow 虚拟环境，如图 2-10 所示。

```
C:\Users\Administrator>activate tensorflow-t
(tensorflow-t) C:\Users\Administrator>
```

图 2-10



执行 `pip install tensorflow --upgrade` 安装 TensorFlow，此时，系统会自动安装 TensorFlow 以及所需的依赖。安装完成后的界面如图 2-11 所示。

```

Downloading http://mirrors.aliyun.com/pypi/packages/71/e1/820d941b5923aac1d49d1c3/e1/60e/361bd290490911bad/7900c19
/ setuptools-39.2.0-py2.py3-none-any.whl (567kB)
100% |#####| 573kB 6.8MB/s
Collecting html5lib==0.9999999 (from tensorboard<1.9.0,>=1.8.0->tensorflow)
Collecting markdown==2.6.8 (from tensorboard<1.9.0,>=1.8.0->tensorflow)
Downloading http://mirrors.aliyun.com/pypi/packages/6d/7d/488b90f470b96531a3f5788cf12a93332f543dbab13c423a5e7ce96a0499
/Markdown-2.6.11-py2.py3-none-any.whl (78kB)
100% |#####| 81kB 10.2MB/s
Collecting bleach==1.5.0 (from tensorboard<1.9.0,>=1.8.0->tensorflow)
Downloading http://mirrors.aliyun.com/pypi/packages/33/70/86c5fec937ea4964184d4d6c4f0b9551564f821e1c3575907639036d9b90
/bleach-1.5.0-py2.py3-none-any.whl
Collecting werkzeug==0.11.10 (from tensorboard<1.9.0,>=1.8.0->tensorflow)
Downloading http://mirrors.aliyun.com/pypi/packages/20/c4/12e3e56473e52375aa29c4764e70d1b8f3efa6682bef8d0aae04fe335243
/ Werkzeug-0.14.1-py2.py3-none-any.whl (322kB)
100% |#####| 327kB 6.4MB/s
Installing collected packages: six, absl-py, gast, astor, setuptools, protobuf, html5lib, numpy, markdown, bleach, werk
zeug, wheel, tensorboard, termcolor, grpcio, tensorflow
Found existing installation: setuptools 36.4.0
Uninstalling setuptools-36.4.0:
Successfully uninstalled setuptools-36.4.0
Found existing installation: wheel 0.29.0
Uninstalling wheel-0.29.0:
Successfully uninstalled wheel-0.29.0
Successfully installed absl-py-0.2.2 astor-0.6.2 bleach-1.5.0 gast-0.2.0 grpcio-1.12.0 html5lib-0.9999999 markdown-2.6.1
1 numpy-1.14.3 protobuf-3.5.2.post1 setuptools-39.2.0 six-1.11.0 tensorboard-1.8.0 tensorflow-1.8.0 termcolor-1.1.0 werk
zeug-0.14.1 wheel-0.31.1
(tensorflow-t) C:\Users\Administrator>

```

图 2-11

如果要安装 GPU 版本的 TensorFlow，只需要将上述的执行命令改为：`pip install tensorflow-gpu --upgrade` 即可，关于 GPU 版本的安装和使用，我们会在后面的章节中讲解。

第三步 运行 TensorFlow

在控制台（CMD）输入 `activate tensorflow-t` 进入 TensorFlow 虚拟环境，然后输入 `python`，进入 python 环境输入代码：

```

>>> import tensorflow as tf
>>> hello = tf.constant('hello Tensorflow!')
>>> sess = tf.Session()
>>> print(sess.run(hello))
>>> b'hello Tensorflow!'

```

如果在控制台成功输出“hello Tensorflow!”，则说明 Tensorflow 运行成功。



小贴士

如果采用 pip 安装 TensorFlow 会出现如图 2-12 所示的警告信息，这是由于使用的是预编译版本进行安装的，不会影响使用，如果不想看到这个提示，可使用源码安装的形式进行安装。

```
>>> sess = tf.Session()
2017-05-30 11:24:39.026497: W c:\tf_jenkins\home\workspace\release-win\device\cpu\windows\tensorflow\core\platform\cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to use SSE instructions, but these are available on your machine and could speed up CPU computations.
2017-05-30 11:24:39.026497: W c:\tf_jenkins\home\workspace\release-win\device\cpu\windows\tensorflow\core\platform\cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to use SSE2 instructions, but these are available on your machine and could speed up CPU computations.
2017-05-30 11:24:39.026497: W c:\tf_jenkins\home\workspace\release-win\device\cpu\windows\tensorflow\core\platform\cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to use SSE3 instructions, but these are available on your machine and could speed up CPU computations.
2017-05-30 11:24:39.026497: W c:\tf_jenkins\home\workspace\release-win\device\cpu\windows\tensorflow\core\platform\cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to use SSE4.1 instructions, but these are available on your machine and could speed up CPU computations.
2017-05-30 11:24:39.026497: W c:\tf_jenkins\home\workspace\release-win\device\cpu\windows\tensorflow\core\platform\cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to use SSE4.2 instructions, but these are available on your machine and could speed up CPU computations.
```

图 2-12

2.1.2 基于 Linux 环境安装 TensorFlow

首先通过 python -V 命令来查看 Linux 下 Python 的版本，如果版本小于 3.X，建议升级到 3.6.X 版本的 Python。

第一步 安装或升级 Python 3.6

(1) 输入：wget https://www.python.org/ftp/python/3.6.2/Python-3.6.2.tgz 命令，下载 python 3.5.2 安装包。

(2) 下载完成后，使用 tar -zxvf Python-3.6.2.tgz 命令进行解压缩。

(3) 通过 cd Python-3.6.2/命令进入 python 目录。

(4) 执行./configure 命令，安装 Python，执行完成后结果如图 2-13 所示。


```

checking for /dev/ptc... no
checking for %zd printf() format support... yes
checking for socklen_t... yes
checking for broken mbstowcs... no
checking for --with-computed-gotos... no value specified
checking whether gcc -pthread supports computed gotos... yes
checking for build directories... done
checking for -O2... yes
checking for glibc _FORTIFY_SOURCE/memmove bug... no
checking for gcc ipa-pure-const bug... no
checking for stdatomic.h... no
checking for GCC >= 4.7 __atomic builtins... yes
checking for ensurepip... upgrade
checking if the dirent structure of a d_type field... yes
checking for the Linux getrandom() syscall... yes
checking for the getrandom() function... no
configure: creating ./config.status
config.status: creating Makefile.pre
config.status: creating Modules/Setup.config
config.status: creating Misc/python.pc
config.status: creating Misc/python-config.sh
config.status: creating Modules/ld_so_aix
config.status: creating pyconfig.h
creating Modules/Setup
creating Modules/Setup.local
creating Makefile

If you want a release build with all stable optimizations active (PGO, etc),
please run ./configure --enable-optimizations

[ root@iZ28hbt7og5Z Python-3.6.2]#
```

图 2-13

(5) 输入 make 命令，编译 Python。

(6) 编译完成后，输入 make install 进行安装，当出现如图 2-14 所示的界面，说明安装完成，不过此时我们的 Python 版本还是系统默认的版本（一般是 2.7.X），此时需要通过 ln -s /usr/local/bin/python3.6 /usr/bin/python 命令进行一下版本替换后即可。

```

Collecting setuptools
Collecting pip
Installing collected packages: setuptools, pip
Successfully installed pip-9.0.1 setuptools-28.8.0
```

图 2-14

(7) 输入 python -V 命令，显示版本为 3.6.2，说明安装 Python 3.6.2 成功，如图 2-15 所示。

```

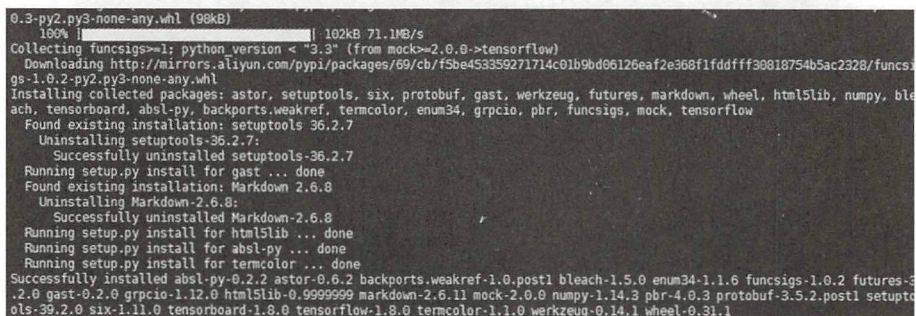
[root@iZ28hbt7og5Z Python-3.6.2]# python -V
Python 3.6.2
[root@iZ28hbt7og5Z Python-3.6.2]#
```

图 2-15



第二步 安装 TensorFlow

接下来，输入 `pip install tensorflow --upgrade` 进行安装。如果看到如图 2-16 所示的界面，则说明安装成功。



```

0.3-py2.py3-none-any.whl (98kB)
100% |#####| 102kB 71.1MB/s
Collecting funcsigs>=1: python_version < "3.3" (from mock>=2.0.0->tensorflow)
  Downloading http://mirrors.aliyun.com/pypi/packages/69/cb/f5be453359271714c01b9bd06126eaf2e368f1fddfff30818754b5ac2328/funcsigs-1.0.2-py2.py3-none-any.whl
Installing collected packages: astor, setuptools, six, protobuf, gast, werkzeug, futures, markdown, wheel, html5lib, numpy, bleach, tensorboard, absl-py, backports.weakref, termcolor, enum34, grpcio, pbr, funcsigs, mock, tensorflow
Found existing installation: setuptools 36.2.7
Uninstalling setuptools-36.2.7:
  Successfully uninstalled setuptools-36.2.7
Running setup.py install for gast ... done
Found existing installation: Markdown 2.6.8
Uninstalling Markdown-2.6.8:
  Successfully uninstalled Markdown-2.6.8
Running setup.py install for html5lib ... done
Running setup.py install for absl-py ... done
Running setup.py install for termcolor ... done
Successfully installed absl-py-0.2.2 astor-0.6.2 backports.weakref-1.0.post1 bleach-1.5.0 enum34-1.1.6 funcsigs-1.0.2 futures-3.2.0 gast-0.2.0 grpcio-1.12.0 html5lib-0.9999999 markdown-2.6.11 mock-2.0.0 numpy-1.14.3 pbr-4.0.3 protobuf-3.5.2.post1 setuptools-39.2.0 six-1.11.0 tensorboard-1.8.0 tensorflow-1.8.0 termcolor-1.1.0 werkzeug-0.14.1 wheel-0.31.1

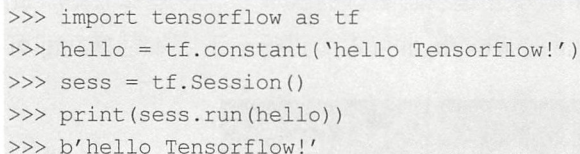
```

图 2-16

如果要安装 GPU 版本的 TensorFlow，只需将命令改为 `pip install tensorflow-gpu --upgrade` 即可。

第三步 运行 TensorFlow

首先输入 `python`，进入 `python` 环境，然后在控制台输入以下代码：



```

>>> import tensorflow as tf
>>> hello = tf.constant('hello Tensorflow!')
>>> sess = tf.Session()
>>> print(sess.run(hello))
>>> b'hello Tensorflow!'

```

如果在控制台成功输出 “hello Tensorflow!”，则说明 Tensorflow 运行成功。

2.2 基于 Java 安装 TensorFlow

TensorFlow 目前也支持使用 Java 语言进行开发，下面将一步步讲解如何基于 Java 语言安装 TensorFlow。

(1) 首先下载 TensorFlow 的 jar 包和 jni 所需要的 dll 文件, 下载地址为: https://www.tensorflow.org/install/install_java, 向下拉动, 下载 jar 包和 jni 文件, 如图 2-17 所示。

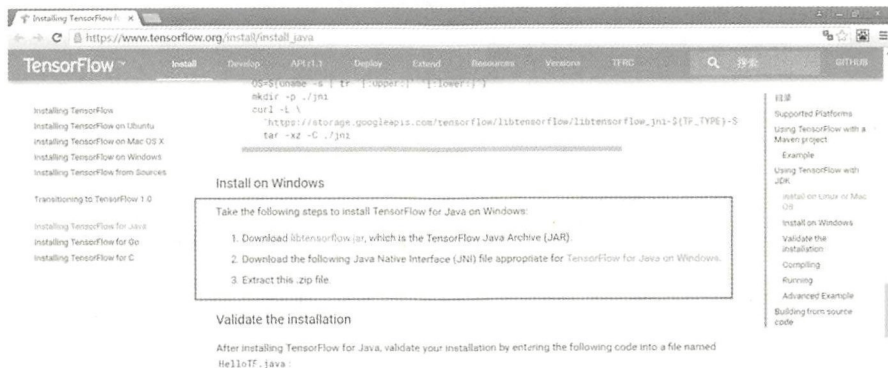


图 2-17

接下来, 在本地使用 Eclipse 新建一个 java 项目, 并将刚刚下载好的 jar 包引入进来, 如图 2-18 所示。

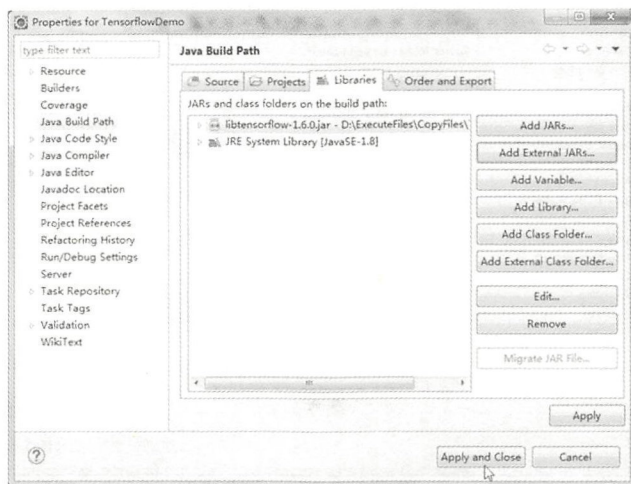


图 2-18

将刚刚下载的 jni 压缩包解压, 将解压后的.dll 文件先放到 libs 目录下(需要自己建立), 如图 2-19 所示。



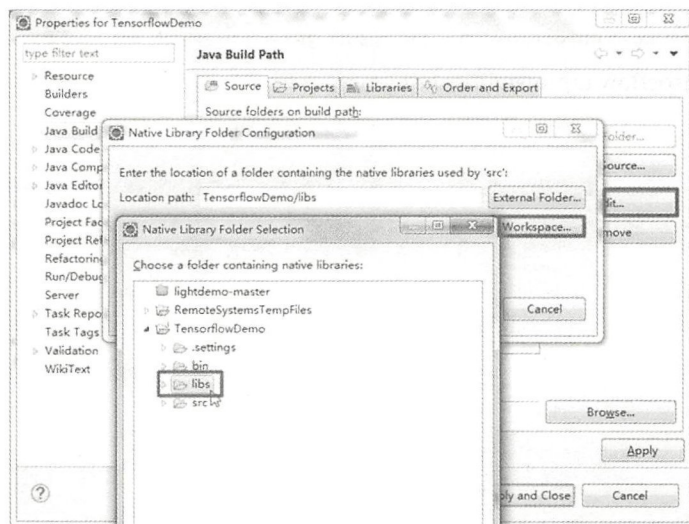


图 2-19

然后引用到工程下，如图 2-20 所示。

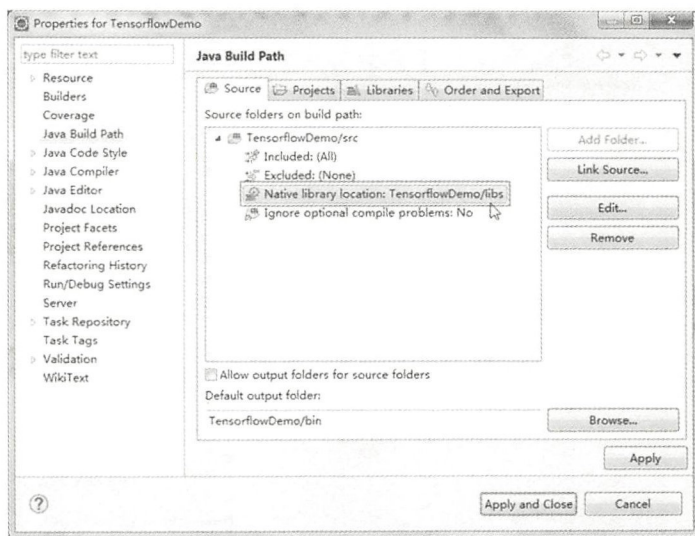


图 2-20

最后完整的目录如图 2-21 所示。



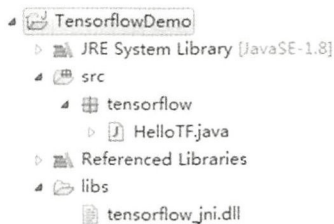


图 2-21

(2) 编写代码:

```
package testTensorflow;

import org.tensorflow.TensorFlow;

public class HelloTensorflow {
    public static void main(String[] args) {

        System.out.println(TensorFlow.version());
    }
}
```

(3) 运行 java 文件, 如果打印出版本名称, 则说明运行成功, 如图 2-22 所示。

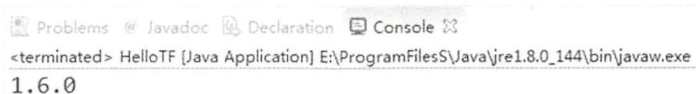


图 2-22

此时, 我们成功将 TensorFlow 使用 Java 运行。

2.3 安装 TensorFlow 的常用依赖模块

在使用 TensorFlow 进行开发时会经常使用一些第三方的依赖库来提高开发效率, 在本书后面的实战部分也会经常用到这些扩展依赖库, 下面来介绍并安装 TensorFlow 所需要的模块。



在 TensorFlow 开发过程中，我们会经常用到以下扩展包。

1. NumPy

NumPy 系统是 Python 开源的数值计算扩展，用来存储和处理大型矩阵，比 Python 自身的嵌套列表（nested list structure）结构要高效得多（该结构也可以用来表示矩阵（matrix））。据说 Python 因 NumPy 而可变身成免费的更强大的 MATLAB 系统。

NumPy 扩展的 pip 安装方法为：

```
pip/pip3 install numpy --upgrade
```

2. SciPy

SciPy 是一款方便、易于使用、专为科学和工程设计的 Python 工具包。它包括统计、优化、整合、线性代数模块、傅立叶变换、信号和图像处理、常微分方程求解器，等等。

SciPy 扩展的 pip 安装方法为：

```
pip/pip3 install scipy --upgrade
```

如果在 Windows 中无法安装，则建议进入官方网站下载安装包，然后执行以下命令进行安装。

```
>python setup.py build  
>python setup.py install
```

3. Matplotlib

Matplotlib 是一个 Python 的 2D 绘图库，它以各种硬拷贝格式和跨平台的交互式环境生成出版质量级别的图形。通过 Matplotlib，开发者仅需要几行代码，便可以生成绘图、直方图、功率谱、条形图、错误图、散点图等。

Matplotlib 扩展的 pip 安装方法为：

```
pip/pip3 install matplotlib --upgrade
```



4. TFLearn

TFLearn 是一个构建在 TensorFlow 之上的模块化和透明的深度学习库，它为 TensorFlow 提供高层次 API，可快速搭建实验环境，同时保持对 TensorFlow 的完全透明和兼容性。后面会详细地讲解 TFLearn 框架。

TFLearn 框架的 pip 安装方法为：

```
pip/pip3 install git+https://github.com/tflearn/tflearn.git
```

5. Librosa

Librosa 库可以很方便地提取音频元素，对音频进行各种处理。

Librosa 库的 pip 安装方法为：

```
pip install librosa --upgrade
```

以上库可使用 Anaconda 进行安装，安装步骤如下所示。

(1) 打开 Anaconda Navigator，切换到 Environments 选项卡，在右侧搜索要安装的扩展，如图 2-23 所示。

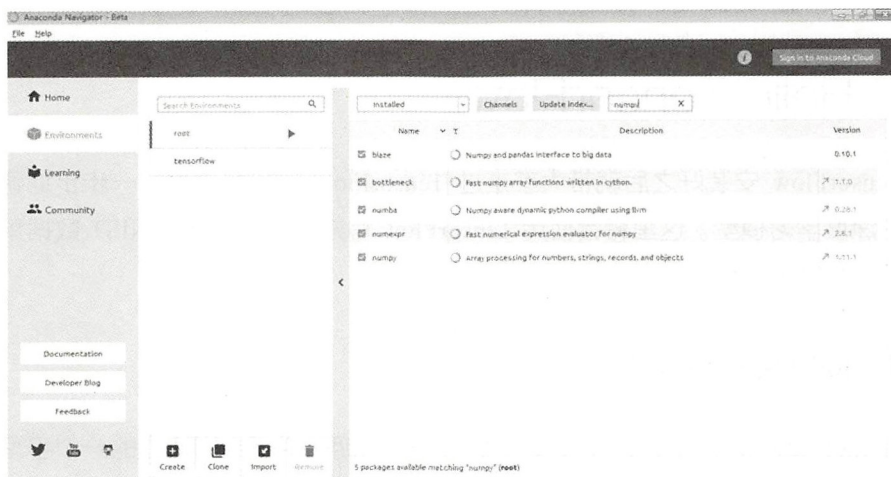


图 2-23

(2) 选择要安装的扩展, 单击 Apply 按钮进行安装, 如图 2-24 所示。

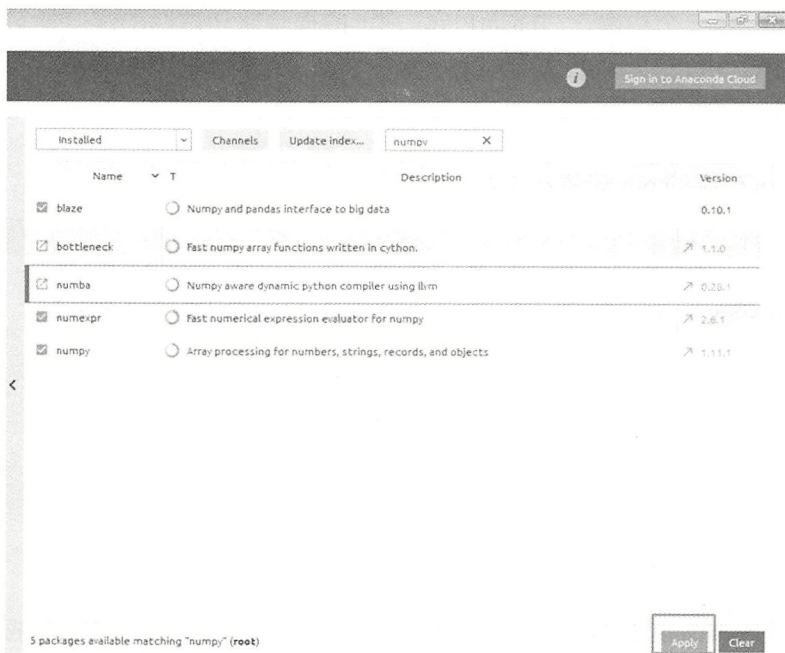


图 2-24

2.4 Hello TensorFlow

TensorFlow 安装好之后就带大家走进 TensorFlow 的代码世界, 一步步地建立起自己的深度学习模型。这里将会使用 TensorFlow 官方最为推荐的 MNIST 数据集实现一个手写数字识别程序。

2.4.1 MNIST 数据集

MNIST 数据集是 Google 实验室和纽约大学柯朗研究所共同建立的一个手写数字的数据库 (一般习惯称为手写数字), 该数据库中共存有 60 000 张手写数字的训练图像和 10 000 张手写数字的测试图像。我们可以在官方网站直接下载数据集, 下载之后会

包含 train-images-idx3-ubyte.gz、train-images-idx3-ubyte.gz、t10k-images-idx3-ubyte. gz、t10k-labels-idx1-ubyte.gz 四个文件，前两个文件是训练数据集的图像和标签，后两个文件是测试数据集的图像和标签。我们所下载的这些图像并不是标准的图像格式（例如 jpg, png 等），而是将它们都保存在二进制文件中，并被转化成一个 28×28 的矩阵。如果通过程序将二进制文件转换成图片，将会得到类似如图 2-25 所示的集合。



图 2-25

其中每一个小的数字如图 2-26 所示。

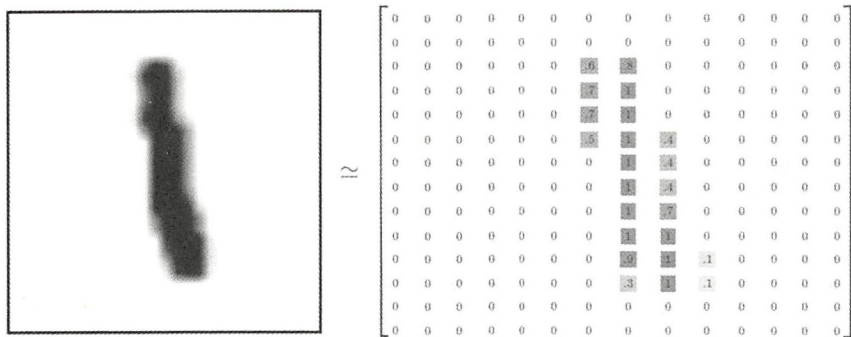


图 2-26

图 2-26 右侧为左侧数字展开的图像，图像总是在矩阵的正中央。

还可以通过脚本文件下载 MNIST 数据集，脚本文件可以在 TensorFlow 目录

\examples\tutorials\mnist 下找到，为了方便，一般建议将其复制一份放在自己工程的目录下。

文件下载完成后，可以编写一个脚本 test_size.py 来查看下载的数据是否正确：

```
import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
print("训练数据的大小为: %d" % (mnist.train.num_examples))
print("测试数据集的大小为: %d" % (mnist.test.num_examples))
```

如果能够正确地输出数据集数量，则说明我们下载的数据集无误：

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

训练数据集的大小为：55 000。

测试数据集的大小为：10 000。

2.4.2 编写训练程序

首先，新建一个项目目录，将官方的 input_data.py 文件放入项目根目录下，并新建一个 mnist.py 文件，然后导入 tensorflow 包和 input_data.py 文件，并使用变量 mnist 来读取数据，在读取数据的过程中，如果发现本地没有相关文件，则会自动下载。

```
import tensorflow as tf
import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

然后定义一个变量 x，并给变量 x 设置一个 placeholder，在 TensorFlow 中，placeholder 一般用来描述一个等待输入的节点，将这个节点设置为 float 类型，并设定其形状（shape）为 [None, 784]，这里的 None 意味着可以是任意的数，一般可以把 placeholder 理解为一个占位符，先将这个位置给预先留出，等到我们后面需要使用的时候，再对其进行赋值操作。

```
x = tf.placeholder("float", [None, 784])
```

接着使用 `tf.Variable()` 方法定义两个变量, `tf.Variable()` 在前面的章节中讲过, 主要为了定义变量, 这里使用 `tf.zeros()` 方法来定义一个形状为 `[784,10]` 的全 0 张量, 也可以理解为一个全 0 矩阵, 这里的 10 一般可以理解为有 10 类, 将其定义为 `w`, 也就是 `weight`, 即权重。

```
w = tf.Variable(tf.zeros([784,10]))
```

再定义一个偏置量, 将其设置为一个 10 维的向量。

```
b = tf.Variable(tf.zeros([10]))
```

下面把向量化后的图片 `x` 和权重矩阵 `w` 相乘, 加上偏置量 `b`, 然后计算每个分类的 Softmax 值。

```
y = tf.nn.softmax(tf.matmul(x,w) +b)
```

接下来定义一个变量 `y_`, `y_` 也是一个二维张量, 其中的每一行为一个 10 维的向量, 用于代表对应某一 MNIST 图片等类别。

```
y_ = tf.placeholder("float", [None, 10])
```

虽然 `placeholder` 的 `shape` 参数是可选的, 但是有了它, TensorFlow 能够自动捕捉因数据维度不一致而导致的错误。

接着, 我们再来定义一个交叉熵, 一般通过交叉熵来描述一个模型的准确程度。

```
cross_entropy = -tf.reduce_sum(y_ * tf.log(y))
```

在这里, 我们通过 `tf.reduce_sum` 函数把 `minibatch` 里的每张图片的交叉熵值进行相加, 计算出的交叉熵就是整个 `minibatch` 的。

下面开始进行模型的训练, 用最基础的梯度下降算法来创建一个优化器 `train_step`。

```
train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)
```

在上面的代码中, 我们将步长设置为 0.01, 这行代码实际上用来往计算图添加一个新的操作, 其中包括了计算的梯度, 以及计算每个参数的步长变化, 并且计算出新的参数值, 最后返回 `train_step` 操作对象, 在运行时会使用梯度下降来更新参数。

接下来调用 `tf.global_variables_initializer()` 方法来初始化创建的变量。

```
init = tf.global_variables_initializer()
```

在有些教材中会使用 `init = tf.initialize_all_variables()` 方法来初始化参数，目前这个方法已经被废弃，以后统一使用 `tf.global_variables_initializer()` 初始化变量。

到目前为止，已经完成了训练 TensorFlow 模型前的所有准备工作，接下来我们需要做以下两件事：

- (1) 训练 TensorFlow 模型；
- (2) 利用测试集来验证训练结果。

在训练 TensorFlow 模型的过程中，一般会使用循环将训练的数据进行多次的训练，以此来保证训练的强度和精度，提高训练的准确率。这里以训练 3 000 次为例，编写训练代码。

```
for i in range(3000):  
    batch_xs, batch_ys = mnist.train.next_batch(100)  
    feed_dict = {x: batch_xs, y_: batch_ys}  
    sess.run(train_step, feed_dict)
```

首先定义了一个 3 000 次的循环，表示训练的次数是 3 000 次，`mnist.train.next_batch(100)` 表示每次迭代都会增加 100 个训练样本数据，然后执行 `train_step` 函数，并通过 `feed_dict` 将 `x` 和 `y` 张量的占位符用训练数据代替。

下一步需要评估训练的精度到底如何。使用 `tf.argmax(y,1)` 的返回值来表示模型对于任一输入 `x` 预测到的标签值，使用 `tf.argmax(y_,1)` 代表正确的标签，使用 `tf.equal` 来检测预测是否与真实的标签匹配，如果索引位置一样则表示匹配。

```
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))  
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
```

最后，我们计算一下所学习到的模型中测试数据集的正确率：

```
print(sess.run(accuracy, feed_dict={x: mnist.test.images,  
y_:mnist.test.labels}))
```


运行以上程序，等待半分钟左右，控制台打印出其预测结果为 0.9148，这个结果每次都是变化的。

除数据外，在进行深度学习模型的构建中，其步长和训练次数也决定了预测的准确率，下面来更改一下程序代码，将步长从 100 调整为 50：

```
batch_xs, batch_ys = mnist.train.next_batch(50)
```

再一次运行程序，查看其预测的正确率，这里打印的结果为 0.9202，那是不是步长越短越好呢？我们来继续做一个实验，将步长再从 50 调整到 10，来看看结果：

```
batch_xs, batch_ys = mnist.train.next_batch(10)
```

运行后，得到的结果为 0.9088，事实证明，并不是步长越小越好。这说明，在深度学习的过程中，任何一个值都是有临界点的，超过了这个临界点，预测精度就会越来越低，因此，找到临界点（在深度学习过程中我们一般将这个过程称之为调参或参数调优），才能使我们的深度学习网络的精确度更高。

本例当中运用的概念和过程在后续章节当中还会有更详细的阐述，如果有不懂的地方不用急，照着做出结果就是胜利。

2.5 小结

本章着重介绍了如何通过 pip 方式、Java 方式和源码方式在 Windows 系统、Linux 系统上安装 CPU 和 GPU 版本的 TensorFlow，以及对在安装过程中可能存在的问题加以说明和解释。接着又介绍了在 TensorFlow 使用过程中会经常用到的扩展模块的安装，这些扩展模块在今后学习的过程中会大量使用。

第 3 章

TensorFlow 基础

本章主要讲解 TensorFlow 的核心基础知识，首先会从 TensorFlow 的系统架构出发，自然地引出张量、计算图、数据流图等 TensorFlow 的核心基础知识，并循序渐进地将其一一讲解。通过学习本章，能够对 TensorFlow 有一个相对深入的了解，便于后续的学习。

3.1 TensorFlow 的系统架构

图 3-1 是一张 TensorFlow 的系统架构图。

TensorFlow 的系统架构以 C API 为界限，将 TensorFlow 分为“前端系统”和“后端系统”两个部分。

前端系统主要是负责提供 TensorFlow 的编程模型，构造计算图和管理 Session 周期。前端可以理解为 TensorFlow 底层给用户暴露出来的 API 接口。开发人员可以使用 C++，Python，Java 等开发语言，在后端系统所提供的 API 接口的基础上根据自身的需求来设计和开发自己所需要的模型，并将模型进行训练，以便日后使用。

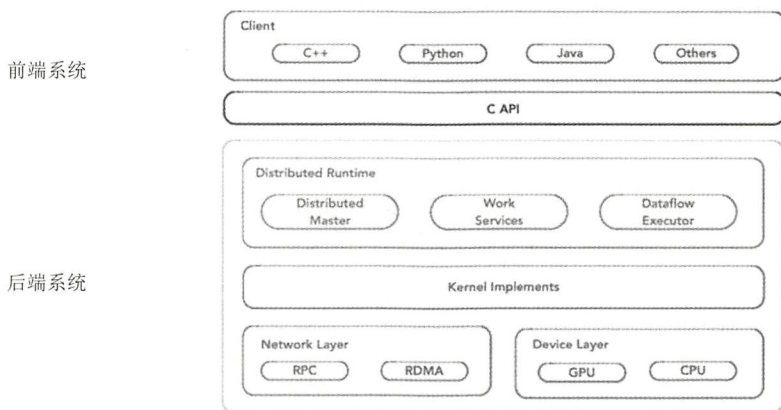


图 3-1 TensorFlow 的系统架构图

后端系统主要提供 TensorFlow 的运行时环境，负责执行计算图，包括计算图的剪枝、设备分配、子图计算等过程。

下面从四个基本组件入手，来着重讲解一下 TensorFlow 的核心系统架构。

3.1.1 Client

Client 是基于 TensorFlow 的编程接口，主要用来构建计算图。一般来讲，我们使用 Python、Java 等编程语言在 TensorFlow 框架上编写好自己的模型后，需要调用 `tf.Session()` 来建立 Client 与后端系统的运行时通道，而且还会触发 Distributed Master 的计算图的执行过程。

以简单的 $y=Wx+b$ 函数为例，来说明 Client 所构建的最简单的计算图，如图 3-2 所示。

图 3-2 首先运用了矩阵的乘法，将 W 与 x 进行矩阵相乘，将相乘所得到的结果，再与 b 相加，最后更新至 y 。

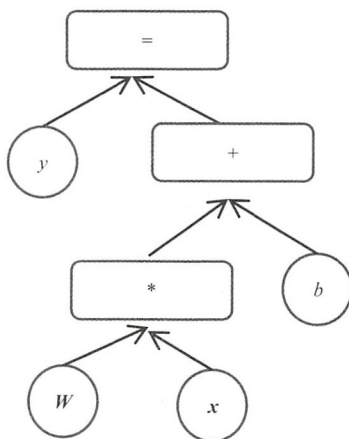


图 3-2

这是线性函数典型的表示方法，其中，一般用 W 来表示权重 weight，用 b 来表示偏差 bias。在今后使用 TensorFlow 进行深度学习开发的过程中，我们会经常遇到这两个值，并用来求解一些基本的线性问题。

将刚刚的 $y=Wx+b$ 再用 Client 流图的形式表示出来，如图 3-3 所示。

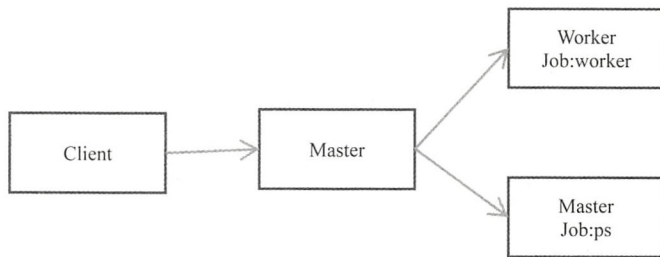


图 3-3

其中，Job:worker 表示计算节点，Job:ps 表示参数保存节点，具体相关知识将在后面的章节详细讲解。

3.1.2 Distributed Master

顾名思义，Distributed Master 主要用于主分布式操作。在分布式的运行环境中

Distributed Master 起到了相当关键的作用。它主要根据 `Session.run()` 方法的 `fetch` 参数，从计算图中反向遍历，找到所依赖的最小的子图，然后将这些子图进行切割操作，切割成多个更小的片段，再利用分布式相关技术和原理，将这些更小的片段分配到不同的进程和设备上进行运算操作，最后 Distributed Master 将这些片段缓存起来，以避免今后的重复操作。

Distributed Master 在开始执行子计算图之前，会先进行一系列的优化操作，目的是使子计算图更加符合我们的计算要求。

3.1.3 Worker Service

Worker Service 是一个工作节点的服务，准确地说，是一个执行部分 TensorFlow 图部分内容的 RPC 服务。对于每一个任务，都会存在一个对应的 Worker Service。

一般来讲，Worker Service 可以最大化地利用 CPU/GPU 的性能，并执行特定的 Kernel。

3.1.4 Kernel Implements

TensorFlow 的内核实现，包含了 200 多个 TensorFlow 运行时的标准的 OP，如状态管理、控制流、多维数组等。

3.2 TensorFlow 的数据结构——张量

顾名思义，TensorFlow 是由 Tensor 和 flow 两部分构成的，其中 Tensor 就是本节要讲的张量（Tensor），而 flow（流）将在后面的章节着重讲解。

3.2.1 什么是张量

张量，在物理学中，是指能够用指标表示法表示的物理量，并且该物理量满足坐标的变换关系。在深度学习领域，可以把张量理解为对神经网络一种高维度的表达方

式，也就是说，张量在神经网络的概念中，可以是任意维度的数据。再简化一些，你也可以变相地直接将张量理解为多维矩阵。

张量具有流动性。张量的流动，指的是在保证计算节点不变的前提下，让数据进行流动。在深度学习的实际操作过程中，会使用大量的算法组成一张图（Graph），张量从图的开始到最后走一遍，叫作完成了一次前向运算；而残差从后向前走一遍，叫作完成了一次向后传播。

3.2.2 张量的阶

在 TensorFlow 中，张量的维度被描述为“阶”，但是，张量的阶和矩阵的阶并不是同一个概念。张量的阶，是对张量维度数量上的描述。例如下面的张量 t 就是一个 2 阶张量：

$$t = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]$$

而下面的张量 m 就是一个 3 阶张量：

$$m = [[[2], [4], [6]], [[8], [10], [12]], [[14], [16], [18]]]$$

一般来讲，我们把 0 阶张量叫作纯量，其特点是只有大小，例如： $s=123$ 就是一个纯量；把 1 阶张量叫作向量，其特点是具有大小和方向，例如 $v=[1.1, 2.2, 3.3]$ 就是一个向量；把 2 阶张量叫作矩阵，也称为数据表，数据表示为 $t=[i,j]$ ，例如 $m=[[1,2,3],[4,5,6],[7,8,9]]$ 就是一个矩阵；而到了 3 阶张量时，一般用来表示一个立体的数据，也就是一个数据体，表示为 $t=[i,j,k]$ ，例如 $t = [[[2], [4], [6]], [[8], [10], [12]], [[14], [16], [18]]]$ 。

3.2.3 张量的形状

TensorFlow 文档中使用了三种记号来方便地描述张量的维度：阶、形状和维数。表 3-1 展示了它们之间的关系。

表 3-1 阶、形状和维数

阶	形 状	维 数	实 例
0	[]	0~ D	一个 0 维张量, 一个纯量
1	[D_0]	1~ D	一个 1 维张量的形式[5]
2	[D_0, D_1]	2~ D	一个 2 维张量的形式[3, 4]
3	[D_0, D_1, D_2]	3~ D	一个 3 维张量的形式[1, 4, 3]
n	[D_0, D_1, \dots, D_n]	$n \sim D$	一个 n 维张量的形式[D_0, D_1, \dots, D_n]

形状可以通过 Python 中的整数、列表或元组 (int、list 或 tuple) 来表示。

3.2.4 数据类型

除了维度, 张量有一个数据类型属性。你可以为一个张量指定下列数据类型中的任意一个类型, 如表 3-2 所示。

表 3-2 数据类型

数据类型	Python 类型	描 述
DT_FLOAT	tf.float32	32 位浮点数
DT_DOUBLE	tf.float64	64 位浮点数
DT_INT64	tf.int64	64 位有符号整型
DT_INT32	tf.int32	32 位有符号整型
DT_INT16	tf.int16	16 位有符号整型
DT_INT8	tf.int8	8 位有符号整型
DT_UINT8	tf.uint8	8 位无符号整型
DT_STRING	tf.string	可变长度的字节数组, 每一个张量元素都是一个字节数组
DT_BOOL	tf.bool	布尔型
DT_COMPLEX64	tf.complex64	由两个 32 位浮点数组成的复数: 实数和虚数
DT_QINT32	tf.qint32	用于量化 Ops 的 32 位有符号整型
DT_QINT8	tf.qint8	用于量化 Ops 的 8 位有符号整型
DT_QUINT8	tf.quint8	用于量化 Ops 的 8 位无符号整型

3.3 TensorFlow 的计算模型——图

3.3.1 计算图基础

在 TensorFlow 中，用图来代表模型的数据流，由 op 和 tensor 组成，其中 op 是操作，也就是节点；而 tensor 是数据流，也就是边。

TensorFlow 通常会被分为两个阶段，即构建阶段和执行阶段。在构建阶段，op 的执行步骤会被描述成一个图；而在执行阶段，则使用会话（Session）来执行图中的 op。

在搭建深度学习的神经网络时，会用到大量的算法和各个图层，具有多层的层级关系。组织各个层级之间关系的这个过程，就是图的构建过程，当图构建完毕的时候，通过不断训练图中的 op 来优化图的各种参数。

在 TensorFlow 中，算法都会被表示成计算图（Computational Graphs），也称为数据流图。可以把计算图看成一种有向图，张量就是通过各种操作在有向图中流动的。

我们继续使用 3.1 节中所讲的 $y=Wx+b$ 来说明问题，如图 3-4 所示。

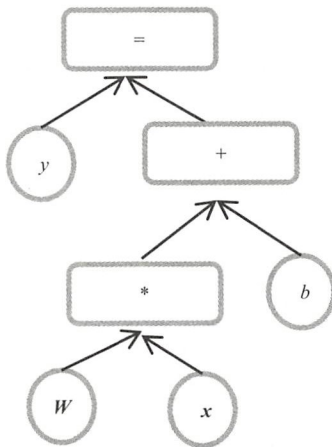


图 3-4

如图 3-4 所示，图中 W ， x ， b ， y 表示图中的四个节点，“+”，“*”，“=” 三个符

号表示三种操作 (op)，图中的节点用一个箭头指向操作符号，这个箭头则可以表示为图的边，在线性函数中， W 也被称为权重 (Weight)，而 b 也被称为偏置量 (bias)，图 3-4 组成了一个完整的线性模型 $y=Wx+b$ 。

3.3.2 计算图的组成

图 3-4 是一个标准而又简单的数据流图，在标准的数据流图中，主要由四个部分构成：

1. 张量 (Tensor)
2. 操作 (Operation)
3. 变量 (Variable)
4. 会话 (Session)

张量的概念在上一节中已经着重讲过，可以把它看作多维矩阵或数组。接下来讲操作、变量和会话。

1. 操作

在进行图计算或者张量的运算中，会经常使用一些方法来对数据进行操作，其中最简单的方法是元素的运算，例如：加 (Add)、乘 (Mul)；有时候还会对矩阵进行运算，例如：矩阵相乘 (MatMul)、矩阵的逆操作 (MatrixInverse) 等；在构建图的时候也避免不了定义一些常量，例如 Constant 操作；在开发深度学习神经网络时，还会用到大量的神经元函数的操作，例如 Softmax、ReLU、Conv2D；最后训练结束了，形成了训练模型，保存时还会用到保存模型的相关操作 Saver, Restore 等。

无论使用哪种操作，都必须对应底层的支持，不管是 CPU 还是 GPU，都要使用不同的操作方式，否则会出现错误提示。

2. 变量

变量就是在计算图的过程中可以改变的节点。比如在如图 3-4 所示的这个例子中，

随着训练次数的增加，权重 W 都会随着迭代的进行而不断地变化着，这样的值就可以拿来作为变量。在实际的训练过程中，一般会把需要训练的值设置为变量。一般来讲，在使用变量的时候，我们都会给变量设置一个初始值，然后根据训练的次数迭代，再将真正的变量不断地推断出来。

一般会通过构造一个 `Variable` 类的实例在图中添加一个变量（`Variable`）。当这个变量被设置初始值后，这个变量的形状和类型就会被固定下来。通过 `assign()` 函数来改变它，通过 `Variable()` 创造的变量可以作为图中的其他操作来使用，你也可以在图中添加节点，对变量进行算术操作。

3. 会话

在 TensorFlow 中，启动一个图的前提是要先创建一个会话（`Session`），TensorFlow 的所有对图的操作，都必须放在会话中进行。在会话中提供了一个 `run` 方法，用来执行计算图的整体或局部的操作。下面举例说明在 TensorFlow 中如何使用会话来执行一个图，以下是一部分代码片段。

```
import tensorflow as tf
Hello = tf.constant('Hello Tensorflow!')
sess = tf.Session()
print(sess.run(Hello))
```

在这个例子中，用 `import` 将 TensorFlow 导入，然后定义了一个内容为“Hello TensorFlow!”的常量并赋值给了 `Hello`，紧接着，通过 `tf.Session()` 方法创建了一个 `session` 对象 `sess`，调用 `session` 对象的 `run` 方法，执行“Hello”，并将其内容打印出来。

调用 `run` 方法执行计算图时，首先要传入一个 `tensor`，这个过程，我们通常称之为填充（`feed`）；最后通过 `sess.run` 方法得到了一个返回结果，我们通常将这个过程称为取回（`fetch`）。

会话（`Session`）在整个计算图执行的过程中充当着桥梁的作用，能够很好地保证计算图执行过程的正确性。关于会话更详细的内容将在 3.4 节中讲解。

3.3.3 计算图的使用

在上一小节中，通过一个非常常见的线性函数的例子讲解了关于计算图的基础知识部分，接下来说一说计算图在实际生产和深度学习领域中的使用，并以代码的形式向大家做一个相对深入的诠释。

先从函数 `tf.Variable()` 说起。

`tf.Variable()` 的作用是添加一个变量到计算图中，在上一小节中也讲到了，`tf.Variable()` 函数必须先经过初始化才能使用。而这个初始值可以是任意的类型或 `shape` 类型的 `Tensor`（关于 `Tensor` 的概念在 3.2 节中有比较详细的讲解）。当使用 `tf.Variable()` 函数构造完成之后，这个变量的 `shape` 就变成了固定的形状，如果想要更改它的值，就必须使用 `assign` 方法，并且使 `validate_shape=False`。

在进行深度学习模型训练的过程中，梯度下降算法、自动求导等问题都是我们一定要面对的，进行图计算的第一步操作就是生成计算图，然后再按照图的执行过程来执行计算图。下面，我们来看一段关于计算图的代码片段：

```
1 import tensorflow as tf
2 A = tf.Variable([[1, 2], [3, 4]], dtype = tf.float32, name='A')
3 B = tf.Variable([[1, 1], [1, 1]], dtype = tf.float32, name='B')
4 y = tf.matmul(A, B)
5 z = tf.sigmoid(y)
6 init_op = tf.global_variables_initializer()
7 with tf.Session() as sess:
8     Sess.run(init_op)
9     z = sess.run(z)
10 print(z)
```

运行后，得到结果：`[[0.95257413,0.95257413],[0.999089,0.999089]]`。

这是一段最简单的计算图的代码，也是本书的第一个比较完整的可以运行的 TensorFlow 源码，下面对这段代码进行一个简单的解释。

第 1 行，在代码中引入 `tensorflow` 库，并设置一个名为 `tf` 的别名；

第 2~3 行，先调用 `tf.Variables()` 函数创建了变量 `A` 和变量 `B`，然后将变量 `A` 初始

化成了一个二维数组,数组的内容为[[1, 2], [3, 4]],并设置了其类型为 `tf.float32` 类型,取了一个别名为“A”;将变量 B 中也初始化成了一个二维数组,数组的内容为[[1, 1], [1, 1]],并设置了其类型为 `tf.float32` 类型,取了一个别名为“B”;

第 4 行,调用了 `tf.matmul()`函数,用来表示两个矩阵或张量相乘,在这里将 A 和 B 相乘,将得到的结果赋值给 y;

第 5 行,调用了 `tf.sigmoid()`函数,并将上一步取得的结果 y 传入到函数中。`tf.sigmoid()`函数是一个非线性函数,主要用来激活上一步取得的结果;

第 6 行,调用了 `tf.global_variables_initializer()`函数,用来初始化前面所有的变量。

第 7~9 行,首先创建了一个会话 session,并为 session 设置了一个别名 sess,然后调用 session 的 `run()`方法,并将刚刚初始化的参数传入,再执行 `run()`方法,将 z 的结果传进去进行训练。

第 10 行,训练结束,打印训练结果。

重要的小贴士:

`tf.global_variables_initializer()`函数是在 2017 年 3 月 2 日开始新加入的函数,此函数替代了原来的 `initialize_all_variables()`函数,目前 `initialize_all_variables()`函数已经被弃用。如果在网上看到有些资料还在用 `initialize_all_variables()`函数,在你使用时,请改成 `tf.global_variables_initializer()`。

`tf.matmul()`函数也是新加入的函数,此函数替代了原来的 `tf.mul()`函数,目前 `tf.mul()`函数已被弃用,如果在网上看到有些资料还在用 `tf.mul()`函数,在你使用时,请改成 `tf.matmul()`。

接下来对上面程序中所用到的一些函数做一个简单的解析:

tf.Variable()函数

```
tf.Variable(initial_value=None, trainable=True, collections=None,
            validate_shape=True, caching_device=None, name=None,
            variable_def=None, dtype=None, expected_shape=None, import_scope=None)
```

这个函数用来定义一个变量，官方给出的参数有 10 种，下面介绍这些参数。

- **initial_value**: 传入值可以是一个张量 Tensor，也可以是可转化为张量的 Python 对象，例如 list，它是变量的初始值。除非将 `validate_shape` 的值设置为 `false`，否则初始值必须具有指定的形状。也可以是没有参数的可调用函数，在调用时返回初始值，在这种情况下，必须指定 `dtype`。
- **trainable**: 如果 `trainable` 为 `True`，则在默认情况下，也会将变量添加到图的集合 `GraphKeys` 中。
- **collections**: 图的键集合。新的变量可以添加到这些集合中去，默认为 `[GraphKeys.GLOBAL_VARIABLES]`。
- **validate_shape**: 如果为 `false`，则允许使用未知形状的初始化变量；如果为 `true`，则使用默认值，但前提是 `initial_value` 的形状必须是已知的。
- **caching_device**: 可选的设备字符串，用于描述缓存变量应该被读取的位置。默认为变量的设备。如果不是 `None`，则可以在其他设备上进行缓存。
- **name**: 变量的可选名称，默认为变量名，并自动获取。
- **Variable_def**: `VariableDef` 协议缓冲区。如果不是 `None`，则使用其内容重新创建 `Variables` 对象，引用该图中的变量节点时，该节点必须已经存在。
- **dtype**: 如果设置此参数，`initial_value` 将被转换为给定的类型。如果为 `None`，数据类型将被保留。
- **expected_shape**: 一个 `tensor` 类型的参数，如果设置此参数，则 `initial_value` 会预先包含这个 `tensor` 的形状。
- **import_scope**: 可选字符串。名称范围添加到变量。仅在协议缓冲区初始化时使用。

tf.matmul()函数

这个函数用来计算矩阵的乘法，将矩阵 a 乘以矩阵 b ，产生矩阵 $a \times b$ 的结果。

`tf.matmul()`函数输入的参数必须是矩阵或二阶以上的张量，并具有匹配的内部尺寸。
`tf.matmul()`函数输入的两个矩阵必须是相同的类型。支持的类型有：`float16`，`float32`，`float64`，`int32`，`complex64`，`complex128`。

`tf.sigmoid()`函数

这个函数用来计算 x 元素的 sigmoid 值。其 sigmoid 的具体表示公式为： $y=1/(1+\exp(-x))$ 。

该函数有两个参数：

- **x**: 类型为 `float32`，`float64`，`int32`，`complex64`，`int64` 或 `int32` 的 Tensor。
- **name**: 操作的名称（可选）。

`tf.global_variables_initializer()`函数

这个函数的作用是初始化全局变量，并返回全局变量的 `op`。

3.3.4 小结

本节重点讲解了 TensorFlow 的计算模型——计算图，并着重讲解了计算图（数据流图）四个重要的构成部分，并通过一个具体的小例子来阐述了计算图的运行过程，以及在计算图运行的过程中所用到的重要函数，并对函数做了介绍，使读者对图的知识有一个全面掌握。

3.4 TensorFlow 中的会话——Session

会话（Session）是运行 TensorFlow 操作的类。在 Session 对象中，封装了执行 `op` 操作的环境，并评估了 Tensor（张量）对象。

会话是图（Graph）和执行者之间的媒介，在使用 `Session.run()`方法运行图时，实际上将 `graph`、`fetch`、`feed_dict` 序列化到字节数组当中去，再将这些操作分发到诸如 CPU 和 GPU 的设备中去，同时提供执行这些 `op` 的方法。当这些方法执行之后，会产

生 Tensor 的返回。在 Python 语言中，返回的 tensor 是 numpy ndarray 对象；在 C 和 C++语言中，返回的 tensor 是 tensorflow::Tensor 实例。

必须在会话里启动一个图，并且，启动图必须在构造完成之后才能够启动。一般来讲，启动图的第一步是创建一个 Session 对象，并对 Session 对象创建参数，否则的话，Session 对象的构造器将会启动一张默认的图。

一般调用 `tf.Session().run()` 方法来启动计算图。在执行计算图的过程中，Session 一般用来输入我们所需要的全部 op。一般来讲，op 在会话中都是并行执行的。

在执行完一个计算图之后，一般会手动地调用 `tf.Session().close()` 函数来关闭会话，将会话进行释放，目的是能够使 CPU/GPU 的使用率达到最大化。当然，如果不手动关闭会话也可以使用 with 代码块来关闭会话，具体代码如下：

```
with tf.Session() as sess:
    result = sess.run([product])
    print result
```

刚才也说过，一般会用 `Session().run()` 方法来启动一个图。但当用 IPython 等之类的 Python 交互环境进行 TensorFlow 开发的时候，可用 `InteractiveSession` 代替 `Session` 类，用 `Tensor.eval()` 和 `Operation.run()` 方法代替 `Session.run()`，以避免使用一个变量来持有会话。



第4章

TensorFlow 中常用的 激活函数与神经网络

4.1 激活函数的概念

我们在刚刚开始接触深度学习的研究时，经常会看到有许许多多的博客、书籍或视频中都会提到一个名词——激活函数。那么激活函数到底是用来干什么的呢？

首先，激活函数并不是真正用来激活什么的，而是指如何把“激活的神经元的特征”通过函数把特征保留并映射出来（保留特征，去除一些数据中的冗余），这是神经网络能解决非线性问题的关键。通俗地说，激活函数的作用是能够给神经网络加入一些非线性因素，使得神经网络可以更好地解决较为复杂的问题。

在激活函数实际应用的过程中，还会涉及如下概念：

饱和

当一个激活函数 $h(x)$ 满足 $\lim_{x \rightarrow +\infty} h'(x) = 0$ 时，我们称之为右饱和。

当一个激活函数 $h(x)$ 满足 $\lim_{x \rightarrow -\infty} h'(x) = 0$ 时，我们称之为左饱和。



当一个函数既满足右饱和的条件又满足左饱和的条件时，我们称为饱和。

硬饱和与软饱和

对任意的 x ，如果存在常数 c ，当 $x > c$ 时恒有 $h'(x)=0$ ，则称其为右硬饱和；当 $x < c$ 时恒有 $h'(x)=0$ ，则称其为左硬饱和。若既满足左硬饱和，又满足右硬饱和，则称这种激活函数为硬饱和。只有在极限状态下偏导数等于 0 的函数才为软饱和。

在对饱和的相关概念有所了解后，接下来就开始真正地进入激活函数的学习当中。

在所有激活函数中，其输入和输出的维度都是一样的。在不同的情况下，应该选用不同的激活函数，才能使深度学习的模型最大程度地发挥价值。

4.2 常用的激活函数

下面就介绍一下在深度学习过程中常用的激活函数。

4.2.1 Sigmoid 函数

Sigmoid 函数也称 S 曲线函数，是使用范围最广的一类激活函数，具有指数函数形状，在物理意义上最为接近生物神经元，在生物学中也是常见的 S 型函数，又称为 S 型生长曲线，是神经网络中最常用的激活函数之一。Sigmoid 函数由下列公式定义：

$$S(x) = \frac{1}{1 + e^{-x}}$$

其函数图像如图 4-1 所示。

Sigmoid 函数由于具有单调递增及反函数单调递增等性质，常被用作神经网络的阈值函数，将变量映射到 0,1 之间。



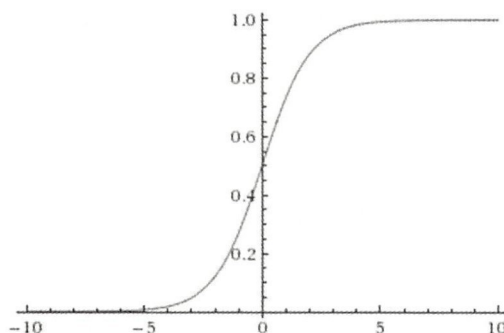


图 4-1

一般来讲，在训练神经网络的过程中，对于求导、连续求导、处理二分类问题，一般使用 Sigmoid 激活函数，因为 Sigmoid 函数可以把实数域光滑地映射到 $(0, 1)$ 空间。函数值恰好可以解释为属于正类的概率（概率的取值范围是 $0 \sim 1$ ）。另外，Sigmoid 函数单调递增，连续可导，导数形式非常简单。但是对于多分类问题，Sigmoid 函数就显得心有余而力不足了。

此外，Sigmoid 函数的输出均大于 0，使得输出不是 0 均值，这称为偏移现象，导致后一层的神经元将得到上一层输出的非 0 均值的信号作为输入。

根据上面的分析，我们来总结一下 Sigmoid 函数的优缺点。

优点

(1) Sigmoid 函数的输出映射在 $(0, 1)$ 之间，单调连续，输出范围有限，优化稳定，可以用作输出层。

(2) 求导容易。

缺点

(1) 由于其软饱和性，容易产生梯度消失，导致训练出现问题。

(2) 其输出并不是以 0 为中心的。

由于在 TensorFlow 框架中已经封装好了 Sigmoid 函数的方法，因此在 TensorFlow



中我们可以直接调用 `tf.sigmoid()` 方法使用 Sigmoid 函数。

下面通过一段代码片段来感受一下 Sigmoid 函数在 TensorFlow 的具体应用。

```
1 import tensorflow as tf
2 A = tf.Variable([[1, 2], [3, 4]], dtype = tf.float32, name='A')
3 B = tf.Variable([[1, 1], [1, 1]], dtype = tf.float32, name='B')
4 y = tf.matmul(A, B)
5 z = tf.sigmoid(y)
```

这段代码相信大家不会陌生,这是第 3 章所用的代码中的一部分,在这段代码中,我们创建了一个矩阵 A,然后又创建了一个矩阵 B,再将矩阵 A 和 B 相乘,相乘后,矩阵的阶就会变得很高,为了方便下一步的使用,调用了 `tf.sigmoid()` 这个函数,对所得的结果 y 使用 Sigmoid 函数进行了激活。

以上是 Sigmoid 函数的代码示例,由此可见,引入此激活函数的主要目的就是为了让深度学习更加方便。

4.2.2 Tanh 函数

Tanh 也是一种非常常见的激活函数。它实际上是 Sigmoid 函数的一种变形。Tanh 函数由下列公式定义:

$$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

无论在中学课本还是百度百科中都会发现 Tanh 函数的公式是:

$$\tanh(x) = \frac{\sinh x}{\cosh x} = \frac{\frac{e^x - e^{-x}}{2}}{\frac{e^x + e^{-x}}{2}} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

那么,现在就由教科书上的公式向深度学习里面的公式做一个推导,让大家彻底地明白是怎么来的:



$$\tanh(x) = \frac{\sinh x}{\cosh x} = \frac{\frac{e^x - e^{-x}}{2}}{\frac{e^x + e^{-x}}{2}} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^x(1 - e^{-2x})}{e^x(1 + e^{-2x})} = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

Tanh 函数的图像如图 4-2 所示。

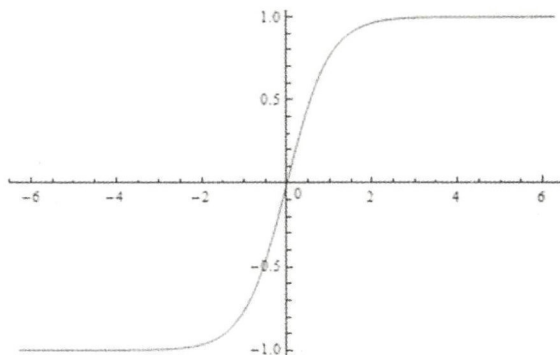


图 4-2

可以把 Tanh 函数看作 Sigmoid 函数的升级版，但是这并不意味着 Tanh 函数在任何情况下都要比 Sigmoid 函数好。

一般来讲，Tanh 函数会在特征相差明显时效果好，在循环过程中会不断扩大特征，效果就显示出来了，但是在特征相差比较复杂或者相差不是特别大时，需要更细微地分类判断的时候，Tanh 函数效果就不太好了。

下面总结一下 Tanh 函数的优缺点。

优点

- (1) Tanh 函数的输出均值是 0，因此收敛速度要比 Sigmoid 函数快。
- (2) 迭代次数相对较少。

缺点

- (1) 与 Sigmoid 函数相同，Tanh 函数同样具有软饱和性。



(2) 还是没有改变 Sigmoid 函数的最大问题——由于饱和性产生的梯度消失。

由于在 TensorFlow 框架中也已经封装好了 Tanh 函数的方法, 因此在 TensorFlow 中可以直接调用 `tf.tanh()` 方法使用 Tanh 函数。

4.2.3 ReLU 函数

ReLU (Rectified Linear Units) 是一种后来才出现的激活函数。与 Sigmoid 函数相比, ReLU 函数的收敛速度会更快, 并且只需要一个阈值就可以得到激活值, 而不是进行一大堆复杂的运算。ReLU 函数的取值是 $\max(0, x)$, 因为神经网络是不断反复计算的, 实际上变成了它在不断试探如何用一个大多数为 0 的矩阵来表达数据特征, 结果因为稀疏特性的存在, 反而使得这种方法运算得又快、效果又好了。ReLU 函数的定义公式是:

$$f(x) = \max(0, x)$$

其函数图像如图 4-3 所示。

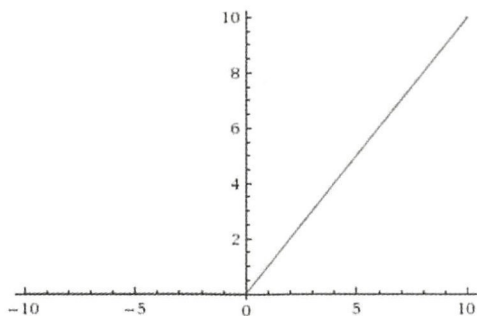


图 4-3

此处需要特别说明的是, 当 $x \leq 0$ 的时候, x 会一直在 x 轴横坐标持续向左, 在 y 轴上永远保持为 0。

可以看到, 当 $x < 0$ 时, ReLU 硬饱和, 而当 $x > 0$ 时, 则不存在饱和问题。所以, ReLU 能够在 $x > 0$ 时保持梯度不衰减, 从而缓解梯度消失问题。这让我们能够直接以监督的方式训练深度神经网络, 而无须依赖无监督的逐层预训练。然而, 随着训练的



推进，部分输入会落入硬饱和区，导致对应权重无法更新。这种现象被称为“神经元死亡”。针对 $x < 0$ 的硬饱和现象，我们一般会对 ReLU 函数做出相应的改进，改进后的函数表达式为：

$$f(x) = \begin{cases} x, & x \geq 0 \\ ax, & x < 0 \end{cases}$$

这里的 a 是一个很小的常数，其存在的目的在于既修正了数据，又保留了部分负轴的值，使得负轴的信息不会全部丢失，这样的变体函数被称为 Leaky-ReLU 函数。其函数图像如图 4-4 所示。

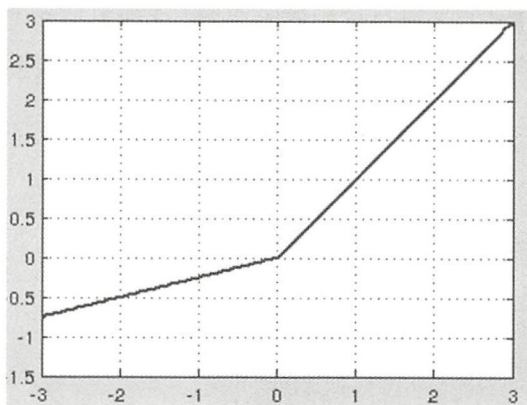


图 4-4

下面总结一下 ReLU 函数的优缺点。

优点

- (1) 相比 Sigmoid 函数和 Tanh 函数，ReLU 函数能够快速收敛。
- (2) Sigmoid 函数和 Tanh 函数涉及很多很高阶的操作（比如指数），ReLU 函数可以更加简单地实现。
- (3) ReLU 函数可以有效地缓解梯度消失问题。
- (4) 在没有无监督预训练的时候也能有较好的表现。



缺点

- (1) 随着训练的进行, 可能会出现神经元死亡、权重无法更新的情况。
- (2) 偏移现象和神经元的死亡会大大影响收敛性。

在 TensorFlow 中, 也封装好了 ReLU 函数, 可以调用 `tf.nn.relu()` 方法来使用 ReLU 函数。

4.2.4 Softplus 函数

Softplus 函数可以看作 ReLU 函数的平滑形式。Softplus 函数也是 Sigmoid 函数的原函数, 也就是说, 如果对 Softplus 函数进行求导的话, 得到的函数就是 Sigmoid 函数。Softplus 函数的表达式为:

$$\text{Softplus}(x) = \log(1 + e^x)$$

其函数图像如图 4-5 所示。

按照有关论文的说法, 一开始想要使用一个指数函数 (天然正数) 作为激活函数来回归, 但是到后期梯度实在太太, 难以训练, 于是加了一个 `log` 来减缓上升趋势。

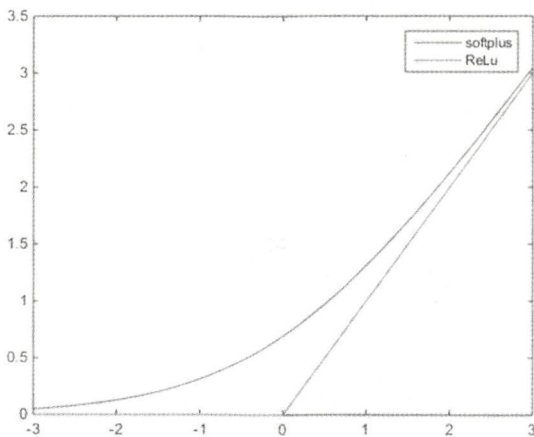


图 4-5



Softplus 函数和 ReLU 函数更加接近脑神经元的激活模型，而神经网络正是基于脑神经科学发展而来的，这两个激活函数的应用促成了神经网络研究的新浪潮。

下面简单列举 Softplus 函数的优点：

(1) 相比 Sigmoid 函数而言，采用 Softplus 函数在进行指数运算时，计算量相对少得多。

(2) 在进行反向传播求误差梯度时，当求导涉及除法，计算量相对少得多。

(3) 对于深层次网络，Softplus 函数不会像 Sigmoid 函数那样很容易就会出现梯度消失的情况，从而会使训练变得更加容易。

在 TensorFlow 框架中，使用 `tf.nn.softplus()` 方法来调用 Softplus 函数。

4.2.5 Softmax 函数

Softmax 函数也是深度学习的常用激活函数，常用于神经网络的最后一层，并作为输出层进行多分类运算。在强化学习中，常用 Softmax 函数作为激活函数，并被用于将某个值转化为激活概率。Softmax 回归模型的函数表达式为：

$$f(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

对于 Softmax 回归模型可以用图 4-6 来解释，对于输入的 x 加权求和，再分别加上一个偏置量，最后再输入到 Softmax 函数中。

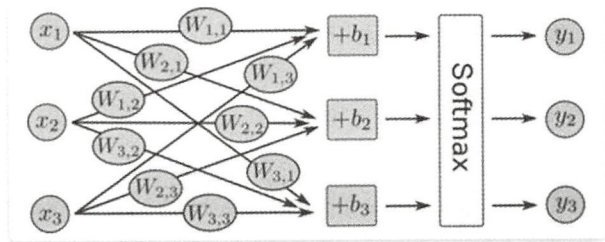


图 4-6



如果把它写成一个等式, 则可以得到 Softmax 数学表达式:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{Softmax} \begin{pmatrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + b_3 \end{pmatrix}$$

也可以用矩阵乘法和向量相加来表示这个计算过程, 有助于提高计算效率(也是一种更有效的思考方式), Softmax 矩阵表达式如下:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{Softmax} \left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

最后, 我们把它写成一个数学公式, 可简化为:

$$y = \text{Softmax}(Wx + b)$$

再拿 Softmax 函数和 Sigmoid 函数做一个对比:

(1) Sigmoid 函数主要针对的是二分类问题, 而 Softmax 函数解决的是多分类问题, 它们的针对点有所不同, 也可以将 Sigmoid 函数看作 Softmax 函数的一个特例。

(2) Softmax 函数基于多项式分布, 而 Sigmoid 函数则基于伯努利分布。

(3) Softmax 函数回归进行多分类时, 类与类之间是互斥的, 而 Sigmoid 函数输出的类别并不是互斥的。

在 TensorFlow 中, 可调用 `tf.nn.softmax()` 方法来实现 Softmax 函数。

4.2.6 小结

本节介绍了在 TensorFlow 使用过程中的几种常用的激活函数, 以及在实际的应用过程中如何选择激活函数进行处理, 通过本节内容, 读者可以对激活函数有一个清晰的认识, 并能够在 TensorFlow 的实际开发中得以实践。

4.3 损失函数的概念

损失函数 (Loss Function) 是一种用来估算预测值和实际值不一样程度的函数, 它是一种非负值函数, 通常使用 $L(Y, f(x))$ 来表示, 一般来讲, 损失函数的值越小, 说明其鲁棒性越好。

一般来讲, 鲁棒性越好, 说明系统越稳定。

卡内基·梅隆大学的 Tom Mitchell 教授在 1998 年给出机器学习的定义: 如果计算机程序对于任务 T 的性能度量 P 通过经验 E 得到了提高, 则认为此程序对 E 进行了学习。按照李航博士的观点, 机器学习包含三要素: 模型、策略和算法, 我们再加上一个要素: 数据, 即构成四个要素。

经验 E 就是指数据, 或者称为数据集, 实际中将数据集分为两部分: 训练样本和测试样本。将其形式化表示为: $D = \{x_i, y_i | 1 \leq i \leq m\}$, m 指数据集的个数, x 是指输入, y 指对应的标签值或者真实值。机器学习, 简单来说就是学习从 x 到 y 之间的映射。如果 y 已经指定好了, 就是有监督学习, 否则就是无监督学习。

模型就是我们所要学习的条件概率分布或者决策函数, 也就是 Tom Mitchell 教授定义中的计算机程序, 模型的假设空间包含所有可能的决策函数, 我们的目的就是模型的假设空间选择最优的一个作为决策函数。直白来说, 假定存在一个函数 f 满足 $y=f(x)$, 指定了函数 f 的形式, 该形式下不同的函数参数取值就构成了模型的假设空间, $F=\{f(x;\theta), \theta \in \Theta\}$, 我们的目的就是求函数 f 的最佳参数。函数 f 可以有不同的形式, 大致可分为广义线性模型和非线性模型, 比如 SVM 属于广义线性模型, 神经网络就是典型非线性模型, 其形式化上可认为是多个非线性函数嵌套, 激活函数就是保证其非线性的关键。

由上可知, 我们要怎么选择最优的函数参数呢? 用什么评价标准呢? 这就是我们讲的要素之一——策略, 也就是性能度量 P 。传统机器学习的策略采用经验风险最小化原则, 要降低经验风险, 就要提高决策函数的复杂度, 但这会导致 VC 维很高。VC 维高, 置信风险就高, 所以结构风险也高, 就产生了过拟合现象, 因此现在使用较多的策略可使结构风险最小化。

什么是经验风险、结构风险和置信风险？首先就要引入本节的主题——损失函数的概念。损失函数是针对单个具体的样本而言的，表示的是模型预测的值 $f(x_i)$ 和样本真实值 y_i 之间的差距，我们用 $L(f(x_i), y)$ 来表示，不同的计算方式就构成不同的损失函数（详见 4.4 节），我们希望的是使这个 L 最小化。通过损失函数，只能知道模型决策函数 f 对单个样本点的预测能力（损失函数越小，说明模型对该样本的预测越准确），那么如果想知道模型 f 对训练样本中所有样本的预测能力，该怎么办呢？显然只需所有的样本求一次损失函数然后累加就好了。如下式：

$$R_{\text{emp}}(f) = \frac{1}{N} \sum_{i=1}^N L(f(X_i), y)$$

这就是经验风险。所谓的经验风险最小化便是让该式子最小化，注意上式中 N 表示的是训练样本中的数量。经验风险是对训练集中的所有样本点损失函数的平均最小化。经验风险越小，说明模型 f 对训练样本的拟合程度越好，但是因为未知样本数据是不确定的，所以就没有办法采用所有样本（包含未知样本和已知的训练样本）损失函数的平均值最小化这个方法。那么怎么来衡量这个样本对所有样本的预测能力呢？熟悉概率论的读者很容易就想到用期望风险，即假设 X 和 Y 服从联合分布 $P(X, Y)$ ，那么期望风险就可以表示为：

$$R_{\text{exp}}(f) = E_p [L(Y, f(x))] = \int L(f(x), y) P(x, y) dx dy$$

这就是期望风险，期望风险是全局的概念，表示的是模型对所有样本预测能力的大小，而经验风险则是局部的概念，仅仅表示模型对训练样本的预测能力。理想的模型（决策函数）应该让所有样本的损失函数最小化（即使期望风险最小化），但是期望风险往往是不可得到的，即上式中， X 与 Y 的联合分布函数不容易得到。现在我们已经清楚了期望风险是全局的，理想情况应该是使期望风险最小化，但是，期望风险又是不容易得到的。怎么办呢？传统机器学习就是采用局部最优代替全局最优的思想，因为如果训练样本的数量足够大，根据大数定理，经验风险应该接近期望风险，这也就是经验风险最小化的理论基础。

但是又有一个问题，如果只考虑经验风险的话，就会出现过拟合的现象，过拟合的极端情况便是模型 $f(x)$ 对训练集中所有样本点都有最好的预测能力，而对于非训练

集中的样本数据，模型的预测能力非常不好。怎么办呢？这时候就引出了结构风险。结构风险是对经验风险和期望风险的折中。结构风险等于经验风险加上置信风险。置信风险是指模型对未知样本进行预测得到的误差。SVM 在小样本训练集上能够得到比其他算法好很多的结果，就在于其优化策略使结构风险最小，而不是使经验风险最小，这样就降低了对数据规模和数据分布的要求。

置信风险的影响因素主要有两个：训练样本数量和模型的 VC 维。训练样本数量越多，置信风险就可以越小，直观理解就是样本多意味着已知经验更丰富，因此遇到新问题出错的风险就低；VC 维越大，模型的假设空间越大，模型解的种类就越多，推广能力就越差，置信风险也就越大。因此，增加训练样本数，降低 VC 维，才能降低置信风险。而模型越复杂意味着 VC 维越高，因此降低模型复杂度就可以降低置信风险。所以一般的做法就是在经验风险后加一个正则化项(惩罚项)便是结构风险了。如下式：

$$R_{\text{svm}}(f) = \frac{1}{N} \sum_{i=1}^N L(f(x_i), y) + \lambda J(f)$$

其中 $J(f)$ 是模型 f 的复杂度， λ 是一个大于 0 的系数，控制惩罚力度，惩罚力度小了即 λ 值较小，或者不惩罚即 λ 为 0，就可能导致过拟合的现象，惩罚力度大了即 λ 值较大可能导致欠拟合，因此 λ 要选取一个适中的值。实际应用中 λ 是一个超参数，需要自行设定，一般通过多次试验得到。具体如图 4-7 所示。

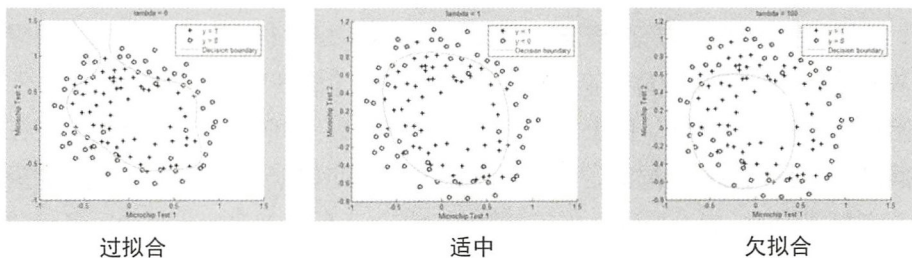


图 4-7

过拟合和欠拟合

我们称学习到的函数和目标函数逼近的吻合程度为拟合。过拟合指模型对训练数

据拟合得太彻底，以至于将训练数据的细节和噪声都学习到了，导致在新的数据上表现很差。欠拟合就是模型没有很好地捕捉到数据特征，不能很好地拟合数据。简单来说：模型在训练数据上的误差较大，为欠拟合；模型在训练数据和测试数据误差均较小，为适中拟合；模型在训练数据集上误差较小，测试集上误差较大，为过拟合。

经验风险越小，模型（决策函数）越复杂，其包含的参数越多，当经验风险小到一定程度就出现了过拟合的现象。当我们想要防止过拟合现象时，就要降低模型（决策函数）的复杂度，即让惩罚项 $J(f)$ 最小化，也就是降低置信风险。为了同时保证经验风险和模型复杂度都达到最小化，简单的办法就是将这两项通过一个权重系数 λ 融合为一个式子，即构成了结构风险函数，然后使这个结构风险函数最小化。

通过以上分析，我们总结一下：

- （1）损失函数度量一次预测的好坏，风险函数度量平均意义下模型的好坏；
- （2）经验风险是局部的，基于训练集所有样本点的损失函数最小化是可求的；
- （3）期望风险是全局的，基于所有样本点的损失函数最小化是不可求的；
- （4）置信风险是基于未知样本点的损失函数最小化，一般通过增加训练样本数量和降低模型复杂度来实现；
- （5）结构风险折中是经验风险加上置信风险，即经验风险加上一个惩罚项。

有了策略之后，也就是有了最优化目标，如何来执行这个策略？即如何计算求解，此时就需要最后一个要素——算法，比如采用结构风险最小化策略后，选定了损失函数的计算形式及惩罚项的形式，意味着要求解 θ 使 $R_{\text{srm}}(f; \theta)$ 最小化，而函数 $R_{\text{srm}}(f; \theta)$ 往往是很复杂的，通常也是非凸的，难以直接计算，因此就需要采用合适的优化算法来求解，比如常用的梯度下降/上升、BP 算法等。

4.4 损失函数的分类

经过上节的理论分析，已经知道了机器学习的策略就是使结构风险最小化，但是

在实际应用中，需要根据任务的实际情况，或者说根据选用的模型，把结构风险最小化的形式表达具体化，得到真正的目标函数。目标函数的一般形式如下所示，也就是说需要确定合适的损失函数计算方式 L ，以及惩罚项的计算方式 Ω ：

$$J = \arg \min_w \sum_i L(y_i, f(x_i; w)) + \lambda \Omega(w)$$

损失函数的定义有多种方式，常见的有均方误差、最大似然估计、最大后验概率和交叉熵损失函数。一般地，选择哪种损失函数，凭经验而定，没什么特定的标准。但是为了便于求解损失函数需要满足两个条件：（1）非负性；（2）当预测值与真实值接近时，损失函数趋近于 0。

具体来说：

（1）均方（平方）误差是一种较早的损失函数定义方法，比较直观，它衡量的是两个分布对应维度的差异性之和；

（2）最大似然估计是从概率的角度出发的，其假设条件是“模型已定，参数未定”，例如，我们知道数据服从正态分布，但是不知道其均值和方差，即可求解出模型参数，使得 $P(y|x, \theta)$ 最大化；

（3）最大化后验概率即使得概率 $P(\theta|x, y)$ 最大化，其假设条件是参数 θ 有一个先验概率，实际上也等价于带正则化项的最大似然估计（读者可自行查询其推导过程），它考虑了先验信息，通过对参数值的大小进行约束来防止“过拟合”；

（4）交叉熵损失函数衡量两个分布 p 、 q 的相似性。在给定集合上两个分布 p 和 q 的交叉熵定义如下：

$$H(p, q) = E_p[-\log q] = H(p) + D_{\text{KL}}(p||q)$$

$$H(p) = - \sum p(x) \log p(x)$$

$$D_{\text{KL}}(p||q) = \sum p(x) \log \frac{p(x)}{q(x)}$$

其中， $H(p)$ 是 p 的熵， $D_{\text{KL}}(p||q)$ 表示 KL 散度（相对熵）。对于离散化的分布 p

和 q :

$$H(p, q) = - \sum_x p(x) \log q(x)$$

在机器学习应用中, p 一般表示训练样本标签的真实分布, 为确定值, 故最小化交叉熵和最小化 KL 散度是等价的, 只不过二者之间相差了一个常数。值得一提的是, 在分类问题中, 交叉熵的本质是对数似然函数的最大化。我们来看一下在 TensorFlow 中交叉熵是如何实现的:

```
cross_entropy=-tf.reduce_mean(y_*tf.log(tf.clip_by_value(y,1e-10,1.0)))
```

其中, $y_$ 代表正确结果, y 代表预测结果。通过 `tf.clip_by_value` 函数可以将一个张量中的数值限制在一个范围内, 避免一些无效的运算, 如 `log0`。`tf.reduce_mean` 是求平均值。

4.5 常用的损失函数

常用的损失函数主要有以下几种。

4.5.1 0-1 损失函数

在分类问题中, 将分类模型 $f(x)$ 作为样本 x 类别的预测值。假设在二分类问题中, 正类 $Y=+1$, 负类 $Y=-1$, 对于一个二分类模型 $f(x)$, 定义 0-1 损失如下:

$$L(Y, f(X)) = \begin{cases} 1, & Y \times f(x) < 0 \\ 0, & Y \times f(x) \geq 0 \end{cases}$$

其等价于下述函数:

$$L(y, f(x)) = \frac{1}{2} (1 - \text{sign}(y \times f(x)))$$

由于 0-1 损失函数只取决于正负号, 是一个非凸的函数, 在求解过程中, 存在很多的不足, 通常在实际应用中使用其替代函数, 如 Log 损失函数。

4.5.2 Log 损失函数

Log 损失函数是 0-1 损失函数的一种替代函数，其形式如下：

$$L(Y, P(Y|X)) = -\log P(Y|X)$$

运用 Log 损失函数的典型分类器是 logistic（逻辑）回归算法。为什么逻辑回归不用平方损失呢？原因在于平方损失函数是线性回归在假设样本是高斯分布的条件下推导得到的（为什么假设高斯分布？其实就是依据中心极限定理）。而逻辑回归的推导中，它假设样本服从于伯努利分布（0-1 分布），然后求得满足该分布的似然函数，接着求取对数等（Log 损失函数中采用 log 就是因为求解过程中使用了似然函数，为了求解方便而添加 log，因为添加 log 并不改变其单调性）。但逻辑回归并没有极大化似然函数，而是转变为最小化负的似然函数，因此有了上式。

已知逻辑函数（Sigmoid 函数）为：

$$h_{\theta}(x) = \frac{1}{1 + e^{-\theta^T x}}$$

可以得到逻辑回归的 Log 损失函数：

$$L(h_{\theta}(x), y) = \begin{cases} -\log(h_{\theta}(x)), y = 1 \\ -\log(1 - h_{\theta}(x)), y = 0 \end{cases}$$

上式的含义就是：如果 $y=1$ ，我们鼓励 $h_{\theta}(x)$ 趋向于 1， $\log(h_{\theta}(x))$ 趋向于 0；如果 $y=0$ ，我们鼓励 $h_{\theta}(x)$ 也趋向于 0， $\log(1 - h_{\theta}(x))$ 也趋向于 0，即满足损失函数的第二个条件，因为 $h_{\theta}(x)$ 小于 1，为了保证损失函数的非负性，即满足第一个条件，所以添加负号。此时将其合并可得单个样本的损失函数：

$$L(h_{\theta}(x), y) = -y_i \log(h_{\theta}(x)) - (1 - y_i) \log(1 - h_{\theta}(x))$$

则全体样本的经验风险函数为：

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m -y_i \log(h_{\theta}(x)) - (1 - y_i) \log(1 - h_{\theta}(x)) \quad \text{式 (1)}$$

该式就是 Sigmoid 函数的交叉熵，这也是上文说的在分类问题上，交叉熵的实质

是对数似然函数。在深度学习中更普遍的做法是将 Softmax 作为最后一层，此时常用的仍是对数似然损失函数，如下所示：

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \sum_{j=1}^k 1\{y_i = j\} \log \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^k e^{\theta_l^T x^{(i)}}} \quad \text{式 (2)}$$

其中 $y_i = j$ 为真时 $1\{y_i = j\} = 1$ ，否则为 0。

该式其实是式 (1) 的推广，正如 Softmax 是 Sigmoid 的多类别推广一样，TensorFlow 根据分类函数 Softmax 和 Sigmoid 就分为 Softmax 交叉熵以及 Sigmoid 的交叉熵，并对这两个功能进行统一封装。

先看 `tf.nn.sigmoid_cross_entropy_with_logits(logits, targets)` 函数，它的实现和之前的交叉熵算法定义是一样的，也是 TensorFlow 最早实现的交叉熵算法。这个函数的输入是 `logist` 和 `targets`，`logist` 就是神经网络模型中的 $W \times X$ 矩阵，注意不需要经过 Sigmoid，因为在函数中会对其进行 Sigmoid 激活，而 `targets` 的 shape 和 `logist` 相同，就是正确的 label 值。

`tf.nn.softmax_cross_entropy_with_logits(logits, targets)` 同样是将 Softmax 和交叉熵计算放到一起了，但是需要注意的是，每个样本只能属于一个类别，即要求分类结果是互斥的，因此该函数只适合单目标的二分类或多分类问题。补充一点，对于多分类问题，例如我们分为 5 类，并且将其人工编码为 0, 1, 2, 3, 4，因为输出值是 5 维的特征，所以需要人工做“one-hot encoding”，即分别编码为 00001, 00010, 00100, 01000, 10000，才能作为该函数的输入。理论上不做“one-hot encoding”也可以，做成和为 1 的概率分布也可以。但一定要保证和为 1，否则 TensorFlow 会检查这些参数，提醒用户更改。

TensorFlow 还提供了一个 `softmax_cross_entropy_with_logits` 的易用版本，即 `tf.nn.sparse_softmax_cross_entropy_with_logits()`，除了输入参数不同外，作用和算法实现都是一样的。`softmax_cross_entropy_with_logits` 的输入必须是类似“one-hot encoding”的多维特征，但像 CIFAR-10、ImageNet 和大部分分类场景都只有一个分类目标，label 值都是从 0 开始编码的整数，每次做“one-hot encoding”比较麻烦，TensorFlow 为了

简化用户操作，在该函数内部高效实现类似“one-hot encoding”，第一个输入函数和前面一样，shape 是 [batch_size, num_classes]，第二个参数以前必须也是 [batch_size, num_classes]，否则无法做交叉熵，而这里将其改为 [batch_size]，但值必须是从 0 开始编码的 int32 或 int64，而且值的范围是 [0, num_class)。如果我们从 1 开始编码或者步长大于 1，则会导致某些 label 值超过范围，代码会直接报错退出。其实，如果用户已经做了“one-hot encoding”，那就可以不使用该函数。

还有一个函数 `tf.nn.weighted_cross_entropy_with_logist()`，是 `sigmoid_cross_entropy_with_logist` 的拓展版，输入和实现两者类似，与后者相比，多支持一个 `pos_weight` 参数，目的是可以增加或减小正样本在算交叉熵时的 loss。

还有一个计算交叉熵的函数，`sequence_loss_by_example(logist, targets, weights)`，用于计算所有 examples（假设一句话有 n 个单词，一个单词及单词所对应的 label 就是一个 example，所有 examples 就是一句话中所有单词）的加权交叉熵损失，logist 的 shape 为 [batch_size, num_decoder_symbols]，返回值是一个 1D float 类型的 tensor，尺寸为 batch_size，其中每一个元素代表当前输入序列 example 的交叉熵。另外，还有一个与之类似的函数 `sequence_loss`，它对 `sequence_loss_by_example` 函数的返回结果进行了一个 `tf.reduce_sum` 运算。

值得一提的是，当最后分类函数是 Sigmoid 和 Softmax 时，不采用平方损失函数。除上文中提到的样本假设分布不同外，还有一个原因是如果采用平方损失函数，则模型权重更新非常慢，假设采用平方损失函数如下式所示：

$$L = \frac{1}{2}(y - \sigma(\mathbf{z}))^2, \text{ 其中 } \mathbf{z} = \mathbf{w}\mathbf{x} + \mathbf{b}$$

采用梯度下降算法调整参数的话，则有

$$\frac{\partial L}{\partial \mathbf{w}} = (y - \sigma(\mathbf{z}))\sigma'(\mathbf{z})\mathbf{x}$$

$$\frac{\partial L}{\partial \mathbf{b}} = (y - \sigma(\mathbf{z}))\sigma'(\mathbf{z})$$

可知 \mathbf{w} 和 \mathbf{b} 的梯度跟激活函数的梯度成正比，但是因为 Sigmoid 的性质，导致 $\sigma'(\mathbf{z})$ 在 \mathbf{z} 取大部分值时都会很小， \mathbf{w} 和 \mathbf{b} 更新也会非常慢，如图 4-8 所示。

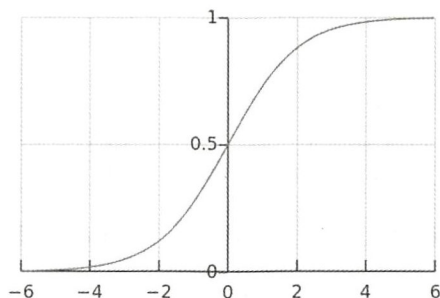


图 4-8

而如果采用交叉熵或者对数损失函数，则参数更新梯度变为：

$$\frac{\partial L}{\partial w_j} = \frac{1}{m} \sum_x (y - \sigma(\mathbf{z})) x_j$$

$$\frac{\partial L}{\partial b} = \frac{1}{m} \sum_x (y - \sigma(\mathbf{z}))$$

可以看到，没有 $\sigma'(\mathbf{z})$ 这一项，权重的更新受误差影响，误差越大权重更新越快，误差越小权重更新就越慢，这是一个很好的性质。

为什么一开始我们说 Log 损失函数也是 0-1 损失函数的一种替代函数，因为 Log 损失函数其实也等价于如下形式：

$$\log\{1 + \exp(-y_i x_i)\}$$

4.5.3 Hinge 损失函数

Hinge 损失函数也是 0-1 函数的替代函数，具体形式如下：

$$L(y, f(x)) = \max(0, 1 - f(x) \cdot y)$$

对可能的输出 $y = \pm 1$ 和分类器预测值 $f(x)$ ，预测值 $f(x)$ 的损失就是上式。运用 Hinge 损失函数的典型分类器是 SVM 算法， $|f(x)| \geq 1$ 。可以看出，当 $f(x)$ 和 y 同符号时，意味着 hinge loss 为 0，但是如果它们的符号相反， $L(y, f(x))$ 则会根据 $f(x)$ 线性增加。

4.5.4 指数损失

具体形式如下：

$$L(y, f(x)) = \exp[-yf(x)]$$

这也是 0-1 函数的一种替代函数，主要用于 AdaBoost 算法。

4.5.5 感知机损失

这也是 0-1 函数的一种替代函数，具体形式如下：

$$L(y, f(x)) = \max(0, -f(x)y)$$

运用感知机损失的典型分类器是感知机算法，感知机算法只需对每个样本判断其是否分类正确，只记录分类错误的样本。与 Hinge 损失函数相比，不同之处在于，Hinge 损失对判定边界附近的点的惩罚力度较高，而感知机损失只要样本的类别判定正确即可，而不需要考虑到边界的距离，这样使其比 Hinge 损失简单，但是泛化能力没有 Hinge 损失强。

这几种损失函数形式如图 4-9 所示，可以看出，除 0-1 函数外，其他函数都可认为是 0-1 函数的替代函数，目的在于使函数更平滑，提高计算性能。

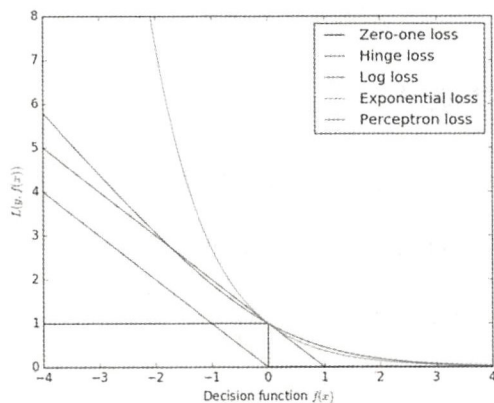


图 4-9

4.5.6 平方（均方）损失函数

具体形式为：

$$L(Y, f(X)) = (Y - f(x))^2$$

平方损失函数较多应用于回归任务，它假设样本和噪声都是服从高斯分布的，是连续的。它的特点是：计算简单方便；欧式距离是一种很好的相似度量标准；在不同的表示域变换后特征性质不变。因此平方损失函数也是一种应用较多的形式。

在 TensorFlow 中计算平方损失，一般采用 `tf.pow(x,y)`，其返回值是 x^y 。举例来说：

```
loss = tf.reduce_mean(tf.pow(y-y_, 2))
```

4.5.7 绝对值损失函数

具体形式为：

$$L(Y, f(X)) = |Y - f(x)|$$

绝对值损失函数与平方损失函数类似，不同之处在于平方损失函数更平滑，计算更简便，因此实际应用中更多地使用平方损失函数。

以上主要讲了损失函数的常见形式，在神经网络中应用较多的是对数损失函数（交叉熵）和平方损失函数。可以看出，损失函数的选择与模型是密切相关的，如果是 square loss，就是最小二乘了，如果是 hinge loss，就是 SVM 了；如果是 exp-loss，那就是 boosting 了；如果是 log loss，那就是 logistic regression 了，等等。不同的损失函数，具有不同的拟合特性，就需要具体问题具体分析。

4.5.8 自定义损失函数

Tensorflow 不仅支持经典的损失函数，还可以优化任意的自定义损失函数。TensorFlow 提供了很多计算函数，基本可以满足自定义损失函数可能会用到的计算操

作。举例来说，预测商品销量时，假设商品成本为 1 元，销售价为 10 元，如果预测少一个，意味着少挣 9 元，但预测多一个，意味着损失 1 元，因为希望利润最大化，所以损失函数不能采用均方误差，需要自定义损失函数，定义如下：

$$L = \sum_{i=1}^n l(f(x), y)$$
$$l(f(x), y) = \begin{cases} a(f(x) - y) & f(x) > y \\ b(y - f(x)) & f(x) \leq y \end{cases}$$

在 TensorFlow 中可以这样定义：其中 `tf.greater()` 用于比较输入两个张量每个元素的大小，并返回比较结果。`Tf.select()` 会根据第一个输入是否为 `true`，来选择第二个参数，还是第三个参数，类似三目运算符。

```
Loss=tf.reduce_sum(tf.select(tf.greater(v1,v2),a*(v1-v2),b*(v2-v1)))
```

4.6 正则项

我们的策略为结构风险最小化，包括经验风险和置信风险，也就是说目标函数对应于两项：第一项为损失函数；第二项则为惩罚项或者叫正则项，该项目的目的在于降低模型复杂度，防止过拟合。如下式所示。

$$J = \arg \min_{\mathbf{w}} \sum_i L(y_i, f(x_i; \mathbf{w})) + \lambda \Omega(\mathbf{w})$$

一般来说，规则项就是模型参数向量 \mathbf{w} 的范数。范数分为：零范数、一范数、二范数、迹范数、Frobenius 范数、核范数，等等，不同的范数对参数 \mathbf{w} 的约束不同，取得的效果也不同。下面我们将选几个常见的范数进行讲解。

4.6.1 L0 范数和 L1 范数

L0 范数， $\|\mathbf{w}\|_0$ ，指向量中非 0 的元素个数。如果我们用 L0 范数来规则化一个参数矩阵 \mathbf{w} 的话，就是指希望 \mathbf{w} 的大部分元素都是 0，也就是使模型“稀疏”。“稀疏”

在神经网络中是一个很好的特性，因为研究表明人脑的神经元连接就是稀疏的，所以好的神经网络也应该是稀疏的。在神经网络的相关资料里，“稀疏”出现的频率也是很高的，比如稀疏自编码等就用到了稀疏性，其实也很好理解，模型稀疏可以过滤没用的特征信息，使特征可以自动选择。但是如果读者去查阅资料可以看到，稀疏性都是通过 L1 范数来实现的，几乎看不到 L0 范数，这是为什么？

L1 范数， $\|W\|_1$ ，是指向量中各个元素的绝对值之和，也称为“稀疏规则算子”（Lasso Regularization）。为什么它会使权值稀疏？一种说法是：它是 L0 范数的最优凸近似。另一种更完备的说法是：对于任何规则化算子，如果它在 $w_i=0$ 的地方不可微，并且可以分解成为一个求和的形式，那么这个规则化算子就可以实现稀疏。其实可以这么理解，目标函数是直接加上模型参数 w 范数并乘以一个权重，我们最小化目标函数，其实就是使 w 的绝对值为 0，这样必然导致模型稀疏。还有一个问题就是为什么不用 L0 范数，而用 L1 呢？笔者认为因为 L0 范数难以求解优化，属于 NP 问题，而 L1 是 L0 的最优凸近似，比 L0 容易求解优化。

总结一下，L0 和 L1 范数都可以实现稀疏，但是由于 L1 比 L0 更易优化求解，因此被广泛应用。

TensorFlow 中 L1 范数采用 `tf.contrib.layers.l1_regularizer()` 实现，输入 `lambda` 为正则化系数， w 为需要正则化的权重，示例如下：

```
loss =  
tf.reduce_mean(tf.square(y-y_)+tf.contrib.layers.l1_regularizer(lambda)(w))
```

4.6.2 L2 范数

L2 范数， $\|W\|_2$ ，也是应用较多的规则化范数，也称为“岭回归”（Ridge Regression），也有人称为“权重衰减”（Weight Decay）。它的主要功能是改善过拟合问题。

L2 范数是指向量各元素的平方和然后求平方根。我们让 L2 范数最小，可以使 w 的每个元素都很小，但与 L1 范数不同，L2 范数使权重都接近于 0。举个例子，0.01 的平方就是 0.0001，当 L2 范数到最小时，权重 w 的实际值只是接近于 0。参数越小则模型越简单，越简单则越不会产生过拟合的现象。

从机器学习理论来看, L2 范数可以防止过拟合, 提升模型的泛化能力; 从优化的角度来说, L2 范数有助于模型优化求解变得稳定和快速。具体细节, 有兴趣的读者可用搜索引擎查询相关资料, 本节不再展开。

从贝叶斯先验的角度看, 加入正则项相当于加入了一种先验, L1 范数相当于加入了一个 Laplacean 先验, L2 范数相当于加入了一个 Gaussian 先验。如图 4-10 所示。

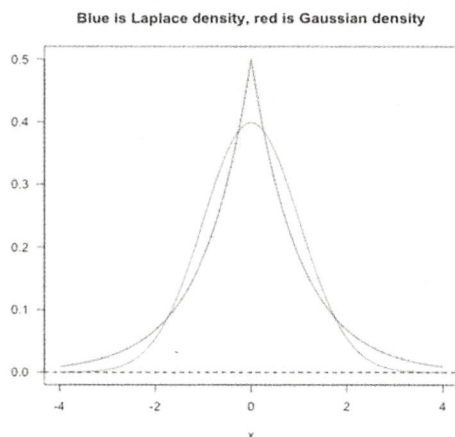


图 4-10

TensorFlow 中 L2 范数与 L1 范数类似, 采用 `tf.contrib.layers.l2_regularizer()` 实现, 输入 `lambda` 为正则化系数, `w` 为需要正则化的权重, 示例如下:

```
loss =  
tf.reduce_mean(tf.square(y-y_)+tf.contrib.layers.l2_regularizer(lambda)(w))
```

4.6.3 核范数

核范数, $\|W\|_*$, 是指矩阵奇异值的和, 它相对于 L1 和 L2 来说, 应用较少, 它的作用是约束低秩 (Low-Rank)。学过矩阵的读者都知道, 秩可以度量矩阵的相关性, 而矩阵的相关性实际上带有矩阵的结构信息。如果矩阵之间各行的相关性很强, 那么就表示这个矩阵实际可以投影到更低维的线性子空间, 也就是用几个向量就可以完全表达了, 此时它就是低秩的。如果用矩阵表达的是结构信息, 例如图像、用户推荐表等, 那么这些矩阵一般是低秩的。

低秩有什么用？如果一个矩阵低秩必然表示其包含大量的冗余信息，利用这种冗余信息，可以对缺失的数据进行恢复，也可以对数据进行特征提取。

那跟核范数又有什么关系？与 L_0 和 L_1 类似，因为矩阵的秩是非凸的，难以直接优化，需要寻找其凸近似，这就是核范数。

核范数主要应用于下面几个方面。

(1) 矩阵填充：矩阵填充常用于推荐系统，先建立不同用户对已购买过的不同商品的评价矩阵，利用低秩重构对矩阵进行填充，从而可以预测用户对未购买过的商品的喜好程度。

(2) 鲁棒 PCA：主成分分析 (PCA) 就是找出数据中最主要的元素和结构，去除噪声和冗余，对数据进行降维，揭示隐藏在复杂数据背后的简单结构。而鲁棒 PCA 考虑的是这样一个问题：一般数据矩阵 X 既包含结构信息，也包含噪声，则可以将这个矩阵分解为两个矩阵相加，一个低秩的（主要的结构信息），另一个是稀疏的（噪声）。优化问题变成如下：

$$\min_{A,E} \|A\|_* + \lambda \|E\|_1 \text{ s.t. } X = A + E$$

比如一副人脸的多幅图像，每幅图像由于受到环境的不同影响，如遮挡、噪声和光照变化等，如果将一副人脸图像拉成一个行向量，多幅人脸图像构成一个矩阵，这个矩阵必然是高度相关的，此时对该矩阵进行低秩和稀疏分解，就可得到干净的人脸图像（低秩矩阵）和噪声矩阵（稀疏矩阵）了。

还可以应用于背景建模，简单情景就是从固定摄像机拍摄的视频中分离背景和前景，可将每一帧图像拉成一个行向量，多副图像则构成一个观测矩阵，由于背景比较稳定，移动物体占据比例较小，可看成噪声，因此进行低秩和稀疏优化可将背景和移动物体分离。

此外还有其他的应用，有兴趣的读者可以深入了解。

4.7 规则化参数

目标函数中除了损失函数和正则项外，还有一个参数 λ ，其实上面也提到，此处再简单说明下，该参数是个超参数，也就是说需要读者自行设定，但是它的设定往往会影响最终模型的好坏，它用于平衡损失函数和正则项对模型参数 W 的影响，一般取一个较小值，如 0.01。读者可以在训练之前，大概计算一下损失函数的值和正则项的值具体是多少，然后根据它们的比例来确定此值，通常需要试验多次，这也是我们常说的调参。

4.8 易混淆的概念

有些初学者可能会混淆：激活函数、分类函数、损失函数、目标函数这几个概念，我们这里再梳理下。

激活函数用于神经网络层与层之间，保证了神经网络的非线性，因此激活函数首选非线性函数。

分类函数应用于分类任务，是网络最后的输出函数，常用 Sigmoid（二分类）和 Softmax（多分类）等，注意到，由于 Sigmoid 函数具有非线性特征，所以以往也常被用作激活函数，但是其会导致梯度消失的现象，现在激活函数一般采用 ReLU 及其变形。

损失函数就是度量模型输出值和真实值之间的差异，对应于经验风险，主要根据网络最后的输出函数的形式来选取，如果网络是分类任务，最后的输出函数（分类函数）采用了 Sigmoid 或者 Softmax，则一般损失函数采用对数损失函数，也就是交叉熵。如果是回归任务，网络最后输出函数是 ReLU 函数，则一般采用平方损失函数。

目标函数是指模型训练的总目标，对应于策略，分为经验风险和结构风险。如果是结构风险，则目标函数包括损失函数和正则项；如果是经验风险则只包括损失函数，此时目标函数和损失函数是等价的。

4.9 神经网络的优化方法

在使用机器学习和深度学习相关技术进行训练时，我们不仅仅要推导出最终的目标函数，针对目标函数进行相关优化也是深度学习中的重要一环。一般来讲，我们会使用梯度下降算法（Gradient Descent）、随机梯度下降算法（Stochastic Gradient Descent）、动量算法（Momentum）等进行神经网络的相关优化。这些算法的使用，不仅简化了训练的过程，还提高了计算的效率，大大地降低了神经网络学习的难度。

下面，我们会重点讲解在 TensorFlow 中如何使用这些算法进行神经网络的优化。

4.9.1 梯度下降算法

梯度下降算法（Gradient Descent）是一种最简单也是最常用的目标函数优化算法，目前大部分深度学习的常用算法都是以梯度学习算法为基础，并在其上面进行拓展而来的，比较常见的衍生算法有随机梯度下降算法（Stochastic Gradient Descent）、批量梯度下降算法（Batch Gradient Descent）、小批量梯度下降算法（Mini Batch Gradient Descent）等。

首先通过一个直观的例子来认识一下梯度下降算法。例如，要求函数 $f(x) = x^2$ 的最小值，如果利用梯度下降算法，则解题步骤如下：

（1）求梯度 $f'(x)$ （我们可以暂时将其看作求导数）， $\nabla = 2x$ 。

（2）将梯度向相反的方向移动 x ， $x - \gamma \cdot \nabla$ ，其中 γ 为步长。如果步长足够小，则可以保证每一次迭代都在减小，但是可能导致收敛得太慢；如果步长太大，则不能保证每一次迭代都在减小，也不能保证收敛。

（3）循环迭代步骤（2），直到 x 值的变化使得 $f(x)$ 在两次差值之间已经达到足够小，例如 0.000000001，也就是说，两次迭代计算出来的 $f(x)$ 基本上没有变化，则说明此时已经达到了最小值，但是要注意的是，此时的最小值一般为局部最小值。

（4）此时，输出 x ，即使函数 $f(x)$ 最小时 x 的取值。



梯度下降算法的相关原理如图 4-11 所示。

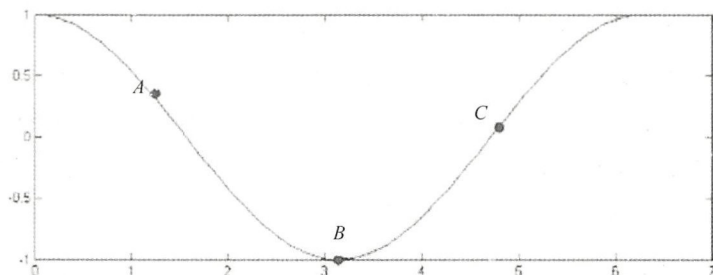


图 4-11

通过图 4-11 可以很直观地发现，假如我现在在 A 点，此时 A 点的偏导数应该是小于 0 的，也就是说 A 点的斜率是小于 0 的；假如我现在在 C 点，此时 C 点的偏导数是大于 0 的，那么也就是说 C 点的斜率是大于 0 的，当 A 点和 C 点的斜率向着 0 的方向进行不断移动的时候，总有一个点 B 会使 A 点和 C 点趋向于无限接近，那么此时我们所得出的这个点就是这个函数的最小值。

那么在整个移动的过程中，又是谁来决定移动的方向呢？这个能够决定移动方向的值，就是我们所说的斜率，也就是前面所提到的步长。随着目标点的移动，斜率 γ 也在不断地发生着变化，当目标函数 $f(x)$ 越来越接近我们想要的值的时候，斜率 γ 变化也会越来越小，此时，我们可以说函数 $f(x)$ 是收敛的；但是如果此时因为其他因素导致斜率 γ 突然变化很大，或者变化很不正常，使得函数 $f(x)$ 值越来越大，越来越远离我们想要得到的结果，此时可以说，函数 $f(x)$ 是发散的。

在 TensorFlow 框架中，我们一般使用 `tf.train.GradientDescentOptimizer()` 函数来进行梯度下降算法的相关优化操作。该函数的主要作用是创建一个梯度下降算法的优化器，其构造函数如下：

```
__init__(
    learning_rate,
    use_locking=False,
    name='GradientDescent'
)
```



在使用的过程中，我们需要传递三个参数：

- **learning_rate**: 需要使用的学习率，一般来讲是使用一个张量或者一个浮点值作为参数。
- **use_locking**: 如果这个值为 True，就对更新操作使用锁。
- **name**: 一个可选的构造器名称，如果为空的话，默认值为 GradientDescent。

下面我们来通过实际的代码来调用一下 `tf.train.GradientDescentOptimizer()` 方法。

```
optimizer = tf.train.GradientDescentOptimizer(0.5).minimize(loss)
```

这是在使用 TensorFlow 的过程中会经常看到的一段话，其主要含义是：

- (1) 调用了 `tf.train.GradientDescentOptimizer()` 方法，定义了一个名为 `optimizer` 的一个优化器。
- (2) 设置其学习率（步长）为 0.5。
- (3) 通过每次的迭代递减让 `loss` 达到最小值。

4.9.2 随机梯度下降算法

前面提过，梯度下降算法是最简单也是最常用的目标函数优化算法，正是由于其简单的特点，导致在处理相对复杂的数据时，会显得有些心有余而力不足。我们来看图 4-12。

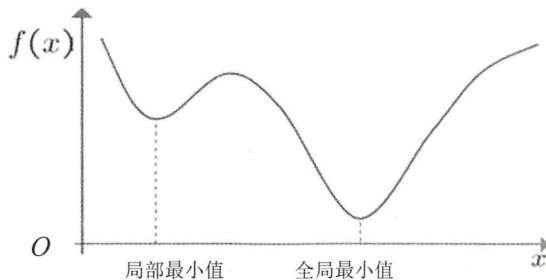


图 4-12

当使用梯度下降算法对其求最小值的时候，梯度下降算法往往是会找在局部范围内的最小值，并不能够保证其得到的最小值是全局性的，如图 4-12 所示，梯度下降算法很有可能会认为 local minimum 是函数 $f(x)$ 的最小值，因此会导致我们最后所得到的结果不是一个最优的结果。

普通梯度下降算法在更新回归系数的时候，一般会遍历整个数据集，但是如果样本的数量巨大，例如有几十万甚至几百万的数据量的时候，那么使用这种方法进行神经网络训练的代价就会变得特别大，耗时会变得特别长，极大地增加了计算的复杂度。这个时候，我们会使用梯度下降算法的一种变体形式——随机梯度下降算法（Stochastic Gradient Descent）进行神经网络的优化。

随机梯度下降算法使用的是近似的方法来减少梯度下降的时间复杂度。

在随机梯度下降算法中，一般会从整个数据集中，随机地选取少量的数据作为数据样本进行计算梯度，并将梯度编号，例如 ∇x_1 ， ∇x_2 ， $\nabla x_3 \dots$ ，我们一般将它们中的每一个称为一个 mini-batch。在随机梯度下降算法中，一般只对一个 mini-batch 进行取值。虽然使用这种方法会使迭代速度变得很快，但是由于每个 mini-batch 的样本量相对较小，所以会使得迭代的次数会大大增加。

4.9.3 其他的优化算法

针对随机梯度下降算法的种种缺点，一般会引入其变形的算法进行深度学习的研究，下面简单地介绍一下。

RMSProp 算法

RMSProp 算法是一种自适应学习率的优化算法，其核心思想是通过统计相似梯度的平均值的方式来自动地调整学习率。一般来讲，我们会在梯度算法中引入一个衰减系数，使每一次双击都有一定的比例。

在 TensorFlow 中，一般使用 `tf.train.RMSPropOptimizer()` 方法来创建一个优化器，`tf.train.RMSPropOptimizer()` 的定义如下：




```
__init__(  
    learning_rate,  
    decay=0.9,  
    momentum=0.0,  
    epsilon=1e-10,  
    use_locking=False,  
    centered=False,  
    name='RMSProp'  
)
```

下面解释这些参数的含义。

- **learning_rate**: 需要使用的学习率，一般来讲是使用一个张量或者一个浮点值作为参数。
- **decay**: 历史或未来的梯度贴现因子，也就是我们刚刚提到的衰减系数。
- **momentum**: 一个梯度的张量。
- **epsilon**: 一个用来避免分母为 0 的值。
- **use_locking**: 如果为 True，则使用锁进行更新操作。
- **Centered**: 如果为 True，则通过梯度的估计方差进行归一化；如果为 False，则由未经过的第二时刻进行归一化。
- **name**: 优化器名称，默认为 RMSProp。

用一段调用代码来说明如何使用 `tf.train.RMSPropOptimizer()` 方法。

```
Optimizer = tf.train.RMSPropOptimizer.__init__(learning_rate, decay=0.9,  
momentum=0.0, epsilon=1e-10, use_locking=False, name=' RMSProp' )
```

在实际应用中，一般会在 RNN 中使用 RMSProp 算法进行优化。

Momentum 算法

Momentum 算法是神经网络的常用优化算法之一，并且也属于梯度下降的变形算法。

首先 Momentum 这个单词译为“动量”，是物理学中很常见的一个名词，在深度学习领域中，Momentum 算法的核心在于在梯度学习算法中加入了动量的概念，使得优化速度会变得更快速。



如果在一个函数中，某些点的方向会相对于其他点的方向向上陡峭得很多，那么使用随机梯度下降算法就会使得函数发生震荡，从而导致发生收敛速度急速下降的现象。为了避免这一现象，我们会在函数中加入动量（Momentum）的概念。从物理学角度来讲，动量可以理解为物体运动时的惯性，也就是说，在我们更新函数中的学习率的时候，会在一定程度上根据之前的学习率进行细微的调整，使其稳定性更强，从而使学习效率变得更高。

在 TensorFlow 中，一般使用 `tf.train.MomentumOptimizer()` 方法来创建一个 momentum 优化器。`tf.train.MomentumOptimizer()`的定义如下：

```
__init__(
    learning_rate,
    momentum,
    use_locking=False,
    name='Momentum',
    use_nesterov=False
)
```

其中，`learning_rate`，`use_locking`，`name` 参数的用法和含义可以参照 `tf.train.RMSPropOptimizer()`函数，其他的函数在此稍作下解析：

- **momentum**：一般用一个张量或者一个浮点型数值表示；
- **use_nesterov**：如果为 `True`，则使用 Nesterov 动量；其中 Nesterov 是为了防止前进太快而做的一个修正，同时也可以提高灵敏度。

Momentum 算法在深度学习领域中的主要作用是提高收敛效率。

Adagrad 算法

普通的梯度下降算法对于所有的参数所使用的学习率都是相同的，但是同一个学习率却不一定适用于所有的参数。相对于梯度下降算法来讲，Adagrad 算法最大的特点就是可以自适应地为各个参数设置不同的学习率。对于各个参数，随着更新总距离的增加，其学习速率也会变慢。

在 TensorFlow 中，一般调用 `tf.train.AdagradOptimizer()`方法来创建一个 Adagrad 优化器。`tf.train.AdagradOptimizer()`的定义如下：



```
__init__(  
    learning_rate,  
    initial_accumulator_value=0.1,  
    use_locking=False,  
    name='Adagrad'  
)
```

其中 `learning_rate`、`use_locking`、`name` 参数可以参照前面的优化器函数，`initial_accumulator_value` 参数表示初始化一个累加器的值，一般来讲我们会将其设置为一个浮点值，并且必须为正数。

Adam 算法

Adam 这个名字来源于自适应矩估计 (Adaptive Moment Estimation)，也是梯度下降算法的一种变形，但是每次迭代参数的学习率都有一定的范围，不会因为梯度很大而导致学习率(步长)变得很大，参数的值相对比较稳定。与 `Adagrad` 函数类似，Adam 函数也是根据每个参数的梯度对每一个参数的学习率进行动态调整的。Adam 优化算法也是在深度学习过程中，经常被用来代替随机梯度下降算法 (SGD) 的首选优化算法。

在 TensorFlow 中，一般调用 `tf.train.AdamOptimizer()` 方法来创建一个 Adam 优化器。`tf.train.AdamOptimizer()` 的定义如下：

```
__init__(  
    learning_rate=0.001,  
    beta1=0.9,  
    beta2=0.999,  
    epsilon=1e-08,  
    use_locking=False,  
    name='Adam'  
)
```

其中 `learning_rate`、`use_locking`、`name` 参数可以参照前面几个优化算法，下面重点解释下其他参数。

- **beta1**: 第一时刻的指数衰减率估计，一般使用一个浮点值或一个常量的浮点张量。
- **beta2**: 第二时刻的指数衰减率估计，一般使用一个浮点值或一个常量的浮点张量。
- **epsilon**: 一个很小的稳定常数，主要用来避免分母为 0 的现象发生。



4.9.4 小结

通过本节的内容，我们掌握了在 TensorFlow 处理神经网络时常用的优化算法，以及如何使用这些优化算法进行优化操作，可将它们应用到具体的代码中。

4.10 生成式对抗网络 (GAN)

我们知道机器学习的方法大致分为两类：生成式方法和判别式方法，分别对应的模型为生成式模型和判别式模型。生成式方法主要对数据样本和标签的联合概率分布 $P(X,Y)$ 进行建模，可用于有监督训练和无监督训练。在有监督任务中通过贝叶斯公式由联合概率分布 $P(X,Y)$ 求出条件概率分布 $P(Y|X)$ ，从而得到预测模型。典型代表有：朴素贝叶斯模型、混合高斯模型和隐马尔科夫模型等。

无监督训练主要学习真实数据集的本质特征，从而刻画样本数据的分布，生成与训练样本相似的新样本数据，可有效地对数据本质特征抽象，典型代表有受限波兹曼机 (RBM)、深度信念网络 (DBN)、深度波兹曼机 (DBM) 和广义除噪自编码器。近两年比较流行的生成式模型主要有三种：生成对抗网络 (GAN)、变分自编码器 (VAE) 和自回归模型。本节将主要介绍生成式对抗网络及其典型变体。

生成式对抗网络 (GAN) 是相对较新的多层网络模型，充分挖掘了多层结构的表达能力，在 2014 年首次提出，核心贡献在于提出了一种新的无监督学习范式，并证明了其可行性。学界大牛 Yann Lecun 曾说，令他最激动的深度学习进展就是生成式对抗网络。

一个典型的生成式对抗网络模型由两个模块或者两个子网络组成，如图 4-13 所示。网络框架主要由生成器 $G(\theta|z)$ 和判别器 $D(x, \theta)$ 组成，其中 z 是随机噪声， x 是真实数据。生成器和判别器均为定义好的多层结构模型。在原始论文里描述为由全连接层构成的多层结构模型。其中判别器的目的是能够分辨出真实数据和来自生成器的生成数据；对应地，生成器的目的是能够生成使得判别器无法正确分辨真实数据和生成数据的数据；在训练过程中固定一方权重，更新另一方网络权重，两个模块交替训练，互相对抗竞争，经过若干步后，直到达到一个纳什均衡，即判别器的正确率为 50%，



此时意味着，生成器生成的数据足以以假乱真，与真实数据基本相当。换句话说，模型的收敛目标是生成器能够从随机噪声生成真实数据。

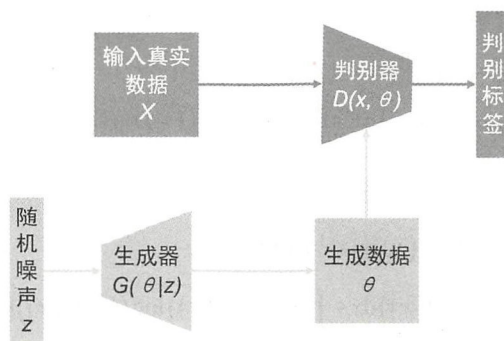


图 4-13¹

$$\min_G \max_D V(D, G) = E_{x \sim p_{\text{data}}(x)} [\log D(x)] + E_{z \sim p_z(z)} [\log(1 - D(G(z)))]$$

上式是生成对抗网络的价值函数，用最大化 $\log D(x)$ 来训练判别器 D 可以最大概率地正确分类生成样本和真实样本，用最小化 $\log(1 - D(G(z)))$ 来训练生成器 G ，最大化判别器的损失。训练过程中固定一方，更新另一方网络参数，交替迭代，使得对方的错误最大化。在 TensorFlow 中，在定义网络时利用 `tf.trainable_variables()` 函数分别获取判别器和生成器的参数，然后将得到的参数列表分别赋给两个优化器的 `var_list` 参数，训练时交替“run”这两个优化器。最终，生成器 G 能估测出样本数据的分布。生成模型 G 隐式定义分布 p_g ，并且希望 p_g 收敛到数据真实分布。当且仅当 $p_g = p_{\text{data}}$ 时存在最优解，即达到纳什均衡，此时生成模型 G 刻画了训练数据的分布，判别器模型 D 的准确率等于 50%。

理论证明

目标证明价值函数 V 可以收敛到最优解，即需要证明最优解存在，以及训练过程收敛。

在训练时，先固定 G 优化 D ，假定：

¹ Goodfellow I, Pouget-Abadie J, Mirza M, et al. Generative adversarial nets[C]//Advances in neural information processing systems. 2014: 2672-2680.



$$D_G^x(x) = \frac{P_{\text{data}}(x)}{P_{\text{data}}(x) + P_g(x)}$$

(1) 证明 $D \times G(x)$ 是最优解, 由于价值函数 V 是连续的, 可以用积分表示期望:

$$\begin{aligned} V(D, G) &= E_{x \sim p_{\text{data}}(x)}[\log D(x)] + E_{z \sim p_z(z)}[\log(1 - D(G(z)))] \\ &= \int_x p_{\text{data}}(x) \log(D(x)) dx + \int_z p_z(z) \log(1 - D(G(z))) dz \\ x = G(z) &\Rightarrow z = G^{-1}(x) \Rightarrow dz = (G^{-1})'(x) dx \\ &\Rightarrow p_g(x) = p_z(G^{-1}(x))(G^{-1})'(x) \\ &= \int_x p_{\text{data}}(x) \log(D(x)) dx + \int_x p_z(G^{-1}(x)) \log(1 - D(x))(G^{-1})'(x) dx \\ &= \int_x p_{\text{data}}(x) \log(D(x)) dx + \int_x p_g(x) \log(1 - D(x)) dx \\ &= \int_x (p_{\text{data}}(x) \log(D(x)) + p_g(x) \log(1 - D(x))) dx \end{aligned}$$

假设 $x=G(z)$ 可逆并进行变量替换, 整理式子得到:

$$\begin{aligned} V(G, D) &= \int_x p_{\text{data}}(x) \log(D(x)) dx + \int_z p_z(z) \log(1 - D(g(z))) dz \\ &= \int_x (p_{\text{data}}(x) \log(D(x)) + p_g(x) \log(1 - D(x))) dx \end{aligned}$$

对价值函数 V 最大化, 即对 D 进行优化使得 V 取最大值:

$$\begin{aligned} \max_D V(D, G) &= \max_D \int_x p_{\text{data}}(x) \log(D(x)) + p_g(x) \log(1 - D(x)) dx \\ \frac{\partial}{\partial D(x)} (p_{\text{data}}(x) \log(D(x)) + p_g(x) \log(1 - D(x))) &= 0 \\ \Rightarrow \frac{p_{\text{data}}(x)}{D(x)} - \frac{p_g(x)}{1 - D(x)} &= 0 \\ \Rightarrow D(x) &= \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)} \end{aligned}$$

通过取极值, 对 V 进行求导并令导数等于 0, 可知 D 的最优解为 $D \times G(x)$ 。

(2) 在最优解为 $D \times G(x)$ 的情况下, G 的最优解为 G 生成的分布与真实分布一致, 即:

$$p_{\text{data}}(x) = p_g(x)$$



在该条件下, $D \times G(x)=1/2$ 。

而 G 的最优解求解则是固定 D , 调整 G :

$$\begin{aligned} C(G) &= \max_D V(G, D) \\ &= \max_D \int_x p_{\text{data}}(x) \log(D(x)) + p_g(x) \log(1-D(x)) dx \end{aligned}$$

将 $D \times G(x)$ 代入, 并去掉前面的 \max :

$$\begin{aligned} &= \int_x p_{\text{data}}(x) \log(D_G^x(x)) + p_g(x) \log(1-D_G^x(x)) dx \\ &= \int_x p_{\text{data}}(x) \log\left(\frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}\right) + p_g(x) \log\left(\frac{p_g(x)}{p_{\text{data}}(x) + p_g(x)}\right) dx \\ &= \int_x p_{\text{data}}(x) \log\left(\frac{p_{\text{data}}(x)}{\frac{p_{\text{data}}(x) + p_g(x)}{2}}\right) + p_g(x) \log\left(\frac{p_g(x)}{\frac{p_{\text{data}}(x) + p_g(x)}{2}}\right) dx - \log(4) \\ &= \text{KL}\left[p_{\text{data}}(x) \parallel \frac{p_{\text{data}}(x) + p_g(x)}{2}\right] + \text{KL}\left[p_g(x) \parallel \frac{p_{\text{data}}(x) + p_g(x)}{2}\right] - \log(4) \end{aligned}$$

整理得到:

$$\begin{aligned} C(G) &= \text{KL}\left[p_{\text{data}}(x) \parallel \frac{p_{\text{data}}(x) + p_g(x)}{2}\right] + \text{KL}\left[p_g(x) \parallel \frac{p_{\text{data}}(x) + p_g(x)}{2}\right] - \log(4) \\ \min_G C(G) &= 0 + 0 - \log(4) = -\log(4) \end{aligned}$$

由于 KL 散度大于等于 0, 所以 C 的最小值是 $-\log(4)$, 且当且仅当:

$$p_{\text{data}}(x) = \frac{p_{\text{data}}(x) + p_g(x)}{2} \Rightarrow p_{\text{data}}(x) = p_g(x)$$

即时成立, 因此当 G 产生的数据和真实数据一致时, C 取得最优解。

算法流程

采用 Minibatch 随机梯度下降训练生成式对抗网络, 其中判别器训练迭代次数 k 是超参数, 经过实验, $k=1$ 是最优选择。

```

for 训练迭代次数 do
  for k 次 do
    从噪声先验分布  $p_g(z)$  中采样  $m$  个噪声样本  $\{z^{(1)}, \dots, z^{(m)}\}$ 
    从真实数据分布  $p_{\text{data}}(x)$  中采样  $m$  个真实样本  $\{x^{(1)}, \dots, x^{(m)}\}$ 
    利用随机梯度上升算法更新判别器:
      
$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m [\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))]$$

    end for
    从噪声先验分布  $p_g(z)$  中采样  $m$  个噪声样本  $\{z^{(1)}, \dots, z^{(m)}\}$ 
    利用随机梯度下降算法更新生成器:
      
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m [\log(1 - D(G(z^{(i)})))]$$

  end for
end for

```

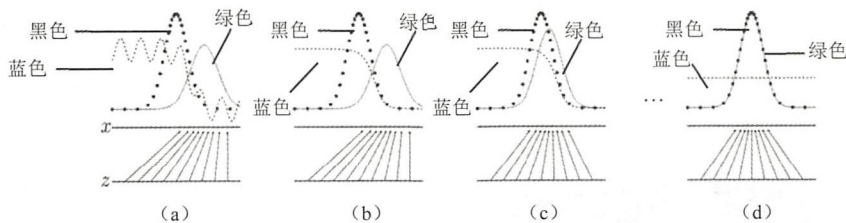
其中基于梯度更新的算法可以采用任一标准梯度更新学习算法，本节采用带有动量的算法。

其中 k 为超参数, k 越大判别器的鉴别能力越强, 最后生成器 G 生成的数据越好, 但需要花费的资源开销越大。

图示化描述

如图 4-14 所示, 蓝色点线代表判别器模型 $D(x)$ 在 x 区域的分布; 黑色点线代表真实数据样本在 x 区域的分布; 绿色点线代表生成器 G 的生成样本在 x 区域的分布; 下面的 $z \rightarrow x$ 表达了生成器 G 通过噪声 z 向混淆数据 x 生成的映射情况。

图 4-14 表明了 GAN 如何一步步学习到真实数据分布的过程, 图 (a) 是训练刚开始, 判别器 D 还不能很好地分辨出 x 分布中的数据来源, $a \rightarrow b$ 是训练算法中的 k 步迭代, 即训练判别器 D , 使其判别能力增强, 到图 (b) 时蓝色点线则比较平滑, 表明判别器 D 已经具有较稳定的判别能力, $b \rightarrow c$ 是训练生成器 G 的过程, 此时判别器 D 的性能固定, 训练 G 使得最小化 $D(G(x))$ 。到图 (c) 时 G 生成的数据逐渐向真实数据靠拢, 生成器 G 对噪声的映射也向中间集中, $c \rightarrow d$ 则是算法多次迭代的过程, 到图 (d) 时, 判别器和生成器达到纳什均衡, 此时生成器 G 已经有足够的混淆能力, 达到最佳性能。判别器 D 此时认为一半数据为真, 一半数据为假。

图 4-14¹

与传统的深度神经网络框架相比，GAN 的优点包括：

- 模型只用到了反向传播，不需要马尔科夫链反复采样，如波兹曼机；
- 训练时不需要对隐变量进行推断，如变分自编码器 VAE，回避了近似计算棘手的概率难题；
- GAN 提供了一种框架，理论上只要是可微分函数都可以用来构建生成器和判别器；
- 生成器的参数更新不是直接来自数据样本，而是来自判别器的反向传播；
- 概率密度不可计算时，仍然可采用 GAN。

GAN 的缺点在于：

- 存在不收敛的问题，当前的理论认为 GAN 应该在纳什均衡上有卓越的表现，但梯度下降只有在凸函数的情况下才能保证实现纳什均衡；
- 可解释性差，生成模型的分布没有显式表达；
- GAN 模型被定义为极大极小问题，没有损失函数，比较难训练，容易出现模式崩溃，生成器容易退化，总是生成同样的样本点，无法继续学习。当生成器崩溃时，判别器也会对相似的样本点指向相似的方向，训练无法继续，因此需要小心平衡生成器和判别器之间训练节拍次数；
- GAN 不需要预定义一个假设分布，即无须预定义模型，会使得模型过于自由不可控。

自提出以来，GAN 引起了众多学者的注意，成为近几年的热点研究领域，原因在于其代表的无监督学习范式有着广阔的前景，从前面可知，当前深度学习的成功主要归功于两点：（1）计算能力提高；（2）大量的训练数据。因此，深度学习要想继续

1 Goodfellow I, Pouget-Abadie J, Mirza M, et al. Generative adversarial nets[C]//Advances in neural information processing systems. 2014: 2672-2680.

发展及更加智能化，必须突破这两个瓶颈，而无监督学习则是一种很好的解决方式。目前生成式对抗网络已经有了许多成功的应用：文字到照片的合成¹；图像超分辨率重建²；图像修复³和纹理合成⁴等，如图 4-15 所示。

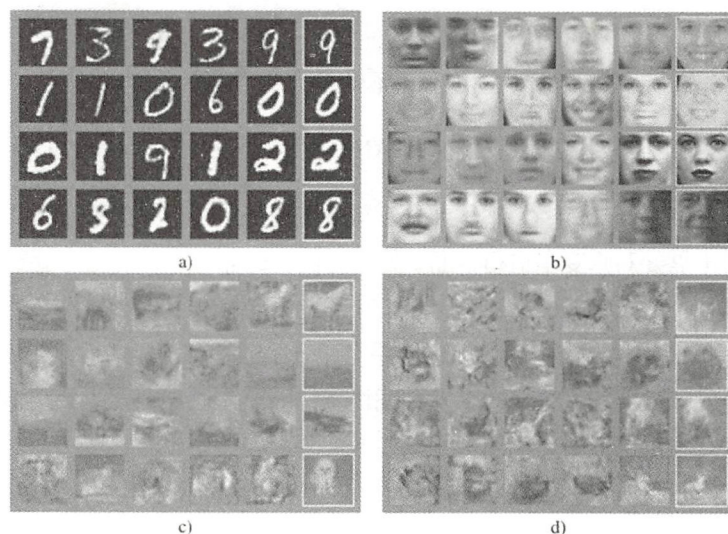


图 4-15⁵

简单的 GAN 实现

```
import tensorflow as tf
import numpy as np
import pickle
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data
```

- 1 Reed S, Akata Z, Yan X, et al. Generative adversarial text to image synthesis[J]. arXiv preprint arXiv:1605.05396, 2016.
- 2 Ledig C, Theis L, Huszár F, et al. Photo-realistic single image super-resolution using a generative adversarial network[J]. arXiv preprint, 2017.
- 3 Li Y, Liu S, Yang J, et al. Generative face completion[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2017, 1(3): 6.
- 4 Zhou Y, Zhu Z, Bai X, et al. Non-Stationary Texture Synthesis by Adversarial Expansion[J]. arXiv preprint arXiv:1805.04487, 2018.
- 5 Mirza M, Osindero S. Conditional generative adversarial nets[J]. arXiv preprint arXiv:1411.1784, 2014.

```

mnist = input_data.read_data_sets('MNIST_data/')
img_size = mnist.train.images[0].shape[0]
noise_size = 100
g_units = 128
d_units = 128
alpha = 0.01
learning_rate = 0.001
smooth = 0.1
batch_size = 128
epochs = 500
n_sample = 25
class GAN():

    def __init__(self, img_size, noise_size):
        self.real_img = tf.placeholder(tf.float32, [None, img_size],
name='real_img')
        self.noise_img = tf.placeholder(tf.float32, [None, noise_size],
name='noise_img')

    @staticmethod
    def get_generator(self, noise_img, n_units, out_dim, reuse=False, alpha=0.01):
        with tf.variable_scope("generator", reuse=reuse):
            # hidden layer
            hidden1 = tf.layers.dense(noise_img, n_units)
            # leaky ReLU
            hidden1 = tf.maximum(alpha * hidden1, hidden1) #Leaky ReLU
            # dropout
            hidden1 = tf.layers.dropout(hidden1, rate=0.2)

            # logist & outputs
            logist = tf.layers.dense(hidden1, out_dim)
            outputs = tf.tanh(logist)
            return logist, outputs

    @staticmethod
    def get_discriminator(self, img, n_units, reuse=False, alpha=0.01):
        with tf.variable_scope("discriminator", reuse=reuse):
            # hidden layer
            hidden1 = tf.layers.dense(img, n_units)
            hidden1 = tf.maximum(alpha * hidden1, hidden1)

            # logist & outputs
            logist = tf.layers.dense(hidden1, 1)
            outputs = tf.sigmoid(logist)

```


TensorFlow 进阶指南

基础、算法与应用

```

        return logist, outputs

    @staticmethod
    def view_samples(self, epoch, samples):
        """
        epoch 代表第几次迭代的图像
        samples 为我们的采样结果
        """
        fig, axes = plt.subplots(figsize=(7,7), nrows=5, ncols=5, sharey=True,
sharex=True)
        for ax, img in zip(axes.flatten(), samples[epoch][1]): # 这里
samples[epoch][1]代表生成的图像结果, 而[0]代表对应的 logist
            ax.xaxis.set_visible(False)
            ax.yaxis.set_visible(False)
            ax.imshow(img.reshape((28,28)), cmap='Grays_r')

        return fig, axes
    def inference(self):

        g_logist, g_outputs = self.get_generator(self, self.noise_img, g_units,
img_size)

        d_logist_real, d_outputs_real =
self.get_discriminator(self, self.real_img, d_units)
        d_logist_fake, d_outputs_fake =
self.get_discriminator(self, g_outputs, d_units, reuse=True)

        self.d_loss_real =
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logist_real,
labels=tf.ones_like(d_logist_real)) * (1 - smooth))

        self.d_loss_fake =
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logist_fake,
labels=tf.zeros_like(d_logist_fake)))
        # 总体 loss
        self.d_loss = tf.add(self.d_loss_real, self.d_loss_fake)

        # generator 的 loss
        self.g_loss =
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logist_fake,

```

```

labels=tf.ones_like(d_logist_fake)) * (1 - smooth))

    train_vars = tf.trainable_variables()
    ## generator 中的 tensor
    self.g_vars = [var for var in train_vars if
var.name.startswith("generator")]
    # discriminator 中的 tensor
    self.d_vars = [var for var in train_vars if
var.name.startswith("discriminator")]
    # optimizer
    d_train_opt = tf.train.AdamOptimizer(learning_rate).minimize(self.d_loss,
var_list=self.d_vars)
    g_train_opt = tf.train.AdamOptimizer(learning_rate).minimize(self.g_loss,
var_list=self.g_vars)

    self.saver = tf.train.Saver(var_list=self.g_vars)

    return d_train_opt,g_train_opt
# 开始训练
def training(self,d_train_opt,g_train_opt):
    samples = []
    losses = []
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        for e in range(epochs):
            for batch_i in range(mnist.train.num_examples//batch_size):
                batch = mnist.train.next_batch(batch_size)

                batch_images = batch[0].reshape((batch_size, 784))
                # 对图像像素进行 scale,这是因为 tanh 输出的结果介于 (-1,1),real 和 fake 图
片共享 discriminator 的参数
                batch_images = batch_images*2 - 1

                # generator 的输入噪声
                batch_noise = np.random.uniform(-1, 1, size=(batch_size,
noise_size))

                # Run optimizers
                sess.run(d_train_opt, feed_dict={self.real_img: batch_images,
self.noise_img: batch_noise})
                sess.run(g_train_opt, feed_dict={self.noise_img: batch_noise})

```


TensorFlow 进阶指南

基础、算法与应用

```

# 每一轮结束计算 loss
train_loss_d = sess.run(self.d_loss,
                        feed_dict = {self.real_img: batch_images,
                                     self.noise_img: batch_noise})

# real img loss
train_loss_d_real = sess.run(self.d_loss_real,
                             feed_dict = {self.real_img: batch_images,
                                             self.noise_img: batch_noise})

# fake img loss
train_loss_d_fake = sess.run(self.d_loss_fake,
                             feed_dict = {self.real_img: batch_images,
                                             self.noise_img: batch_noise})

# generator loss
train_loss_g = sess.run(self.g_loss,
                        feed_dict = {self.noise_img: batch_noise})

if e%100==0:

    print("Epoch {}/{}\...".format(e+1, epochs),
          "Discriminator Loss: {:.4f}(Real: {:.4f) + Fake: {:.4f})..."
          .format(train_loss_d, train_loss_d_real, train_loss_d_fake),
          "Generator Loss: {:.4f}".format(train_loss_g))

    # 记录各类 loss 值
    losses.append((train_loss_d, train_loss_d_real, train_loss_d_fake,
                  train_loss_g))

    # 抽取样本后期进行观察
    sample_noise = np.random.uniform(-1, 1, size=(n_sample, noise_size))
    gen_samples = sess.run(self.get_generator(self, self.noise_img,
                                              g_units, img_size, reuse=True),
                          feed_dict={self.noise_img: sample_noise})
    samples.append(gen_samples)

    self.saver.save(sess, './checkpoints/generator.ckpt')

    with open('train_samples.pkl', 'wb') as f:
        pickle.dump(samples, f)
    return losses

def draw_loss(self, losses):

```

```

fig, ax = plt.subplots(figsize=(20,7))
losses = np.array(losses)
plt.plot(losses.T[0], label='Discriminator Total Loss')
plt.plot(losses.T[1], label='Discriminator Real Loss')
plt.plot(losses.T[2], label='Discriminator Fake Loss')
plt.plot(losses.T[3], label='Generator')
plt.title("Training Losses")
plt.legend()

def draw_samples(self):

    epoch_idx = [0, 5, 10, 20, 40, 60, 80, 100, 150, 250]
    # 一共 300 轮, 不要越界
    show_imgs = []
    with open('train_samples.pkl', 'rb') as f:
        samples = pickle.load(f)
    for i in epoch_idx:
        show_imgs.append(samples[i][1])

    # 指定图片形状
    rows, cols = 10, 25
    fig, axes = plt.subplots(figsize=(30,12), nrows=rows, ncols=cols,
sharex=True, sharey=True)

    for sample, ax_row in zip(show_imgs, axes):
        for img, ax in zip(sample, ax_row):
            ax.imshow(img.reshape((28,28)), cmap='Grays_r')
            ax.xaxis.set_visible(False)
            ax.yaxis.set_visible(False)

def test(self):

    with tf.Session() as sess:
        self.saver.restore(sess, tf.train.latest_checkpoint('checkpoints'))
        sample_noise = np.random.uniform(-1, 1, size=(25, 100))
        gen_samples = sess.run(self.get_generator(self,self.noise_img, g_units,
img_size, reuse=True),
                                feed_dict={self.noise_img: sample_noise})

        self.view_samples(self,0, [gen_samples])

gan=GAN(img_size=784,noise_size=100)

```



```
#训练...
d_train_opt,g_train_opt=gan.inference()
losses=gan.training(d_train_opt,g_train_opt)
gan.draw_loss(losses)
#测试...
gan.draw_samples()
gan.test()
```

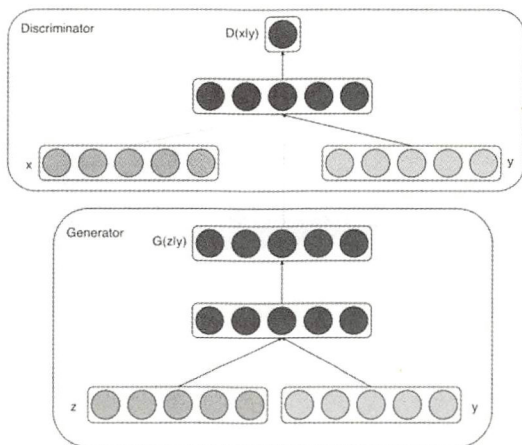
4.10.1 CGAN

在原始 GAN 中，目的是使得生成器能够从随机噪声中生成真实数据，而 CGAN 则更近一层，即给 GAN 加上条件，指导数据的生成过程，使得生成具有特定性质的样本。以生成 MNIST 数据集的图像样本来说，原始 GAN 得到的生成器可以由随机向量生成一张含有数字的图像样本，其中数字可能是 0~9 中的任意一个，而 CGAN 则是在生成器输入时添加一个条件 y ，使得可以生成符合预期数字的图像样本，如生成含有数字 1 的图像。如图 4-16 所示。价值函数变化如下：

$$\min_G \max_D V(D, G) = E_{x \sim P_{\text{data}}(x)} [\log D(x|y)] + E_{z \sim P_Z(z)} [\log(1 - D(G(z|y)))]$$

这里有两组试验¹：一组是 MNIST 手写字体数据生成，一组是 Flickr 数据集的多模态图像自动标注。在 MNIST 实验中，将 label 的 one-hot 编码作为条件变量 y 与 100 维的均匀分布噪声向量一起输入 concat 作为生成器输入，生成对应的数字，最终通过 Sigmoid 输出 784 维的生成数据；在判别器中，采用了 maxout 激活层。在 Flickr 实验中，利用图像的特征作为条件变量 y ，生成次向量分布，实现图像自动标注的功能。首先采用在 ImageNet 数据集上预训练的卷积神经网络作为特征提取器，将其最后的全连接层输出作为图像特征，然后基于 YFCC100M 数据集训练一个 Skip-Gram 模型，作为生成语义的字典，训练过程中分别使用预训练的卷积神经网络和 Skip-Gram 模型来提取 Flickr 数据中图像特征和标签特征，然后用于训练 CGAN，最终实现较好的图像自动标注的效果。

¹ Arjovsky M, Chintala S, Bottou L. Wasserstein gan[J]. arXiv preprint arXiv:1701.07875, 2017.

图 4-16¹

4.10.2 DCGAN

DCGAN 是应用比较广泛的改进结构，基本采用卷积层替代了原始的全连接层，其中在生成器中采用带步长的卷积代替了上采样，极大地提升了 GAN 训练时的稳定性及生成结果质量。如图 4-17 所示。

GAN 的主要问题是训练过程不稳定，而 DCGAN 改进了其稳定性，原因在于：

(1) 几乎每层都使用 batchnorm 层，将特征层的输出归一化到一起，加速训练，提升训练的稳定性；

(2) 判别器中使用 Leaky ReLU，防止梯度过度稀疏，生成器则仍采用 ReLU，但最后输出层采用 Tanh；

(3) 使用 Adam 优化器训练，且最佳学习率为 0.0002；

(4) 使用带步长卷积代替上采样层，卷积在提取图像特征上有较好的作用，并且使用卷积代替全连接层。

¹ Radford A, Metz L, Chintala S. Unsupervised representation learning with deep convolutional generative adversarial networks[J]. arXiv preprint arXiv:1511.06434, 2015.

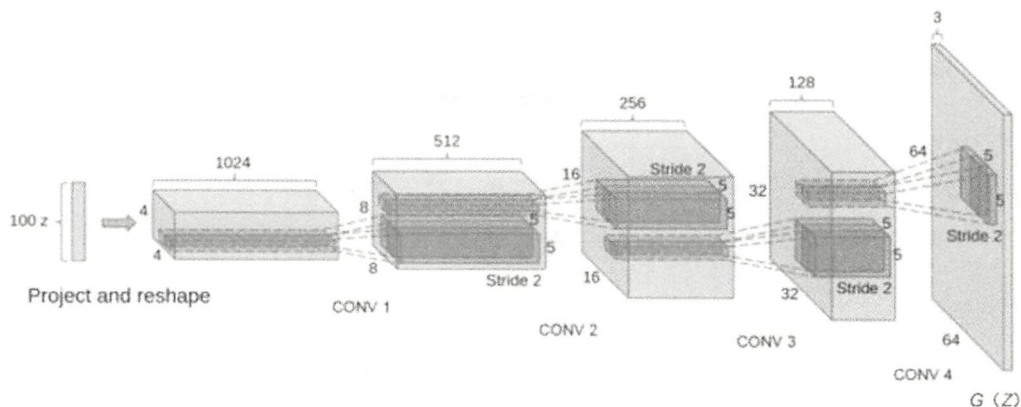


图 4-17

4.10.3 WGAN

为了使得 GAN 的训练更加稳定，与 DCGAN 不同的是，WGAN 主要从损失函数的角度进行改进：

- 判别器最后一层去掉 Sigmoid；
- 生成器和判别器的 loss 不取 Log；
- 对更新后的权重强制 clip，如 $[-0.01, 0.01]$ ，以满足连续性条件；
- 推荐 SGD，RMSProp 等优化器，不要采用含有动量的优化算法，如 Adam。

原始的 GAN 存在的问题有：判别器越好，生成器梯度消失越严重，生成器 loss 降不下去；判别器不好，生成器梯度不准，训练不稳定，只有判别器训练得不好不坏才行，但这个尺度很难把握，甚至同一轮训练的不同阶段该尺度都不一样，所以 GAN 才难以训练。最小化生成器 loss 函数，会等价于最小化一个不合理的距离度量，使得最小化生成分布与真实分布的 KL 散度的同时又要最大化两者的 JS 散度，导致梯度不稳定，同时也会使得生成器宁可多生成一些重复但较为“安全”的样本，也不愿生成多样性的样本，从而导致模式崩溃，即多样性不足。

图 4-18 所示为标准 GAN 和 WGAN 对真实样本分布和生成样本分布判别的差异，标准 GAN 会出现梯度消失的情况，而 WGAN 则有较好的线性梯度。

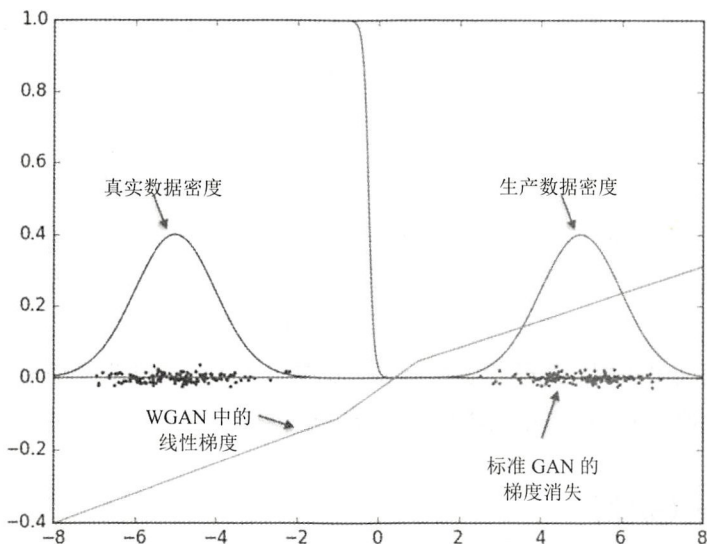


图 4-18¹

WGAN 的贡献主要在于从理论上给出了 GAN 训练不稳定的原因，即交叉熵不适合衡量具有不相交部分的分布之间的距离，转而使用 Wasserstein 距离去衡量生成数据与真实数据分布之间的距离，理论上解决了训练不稳定的问题；解决了模式崩溃问题，生成结果更加多样；对 GAN 的训练提供了一个指标，可以采用此指标来衡量 GAN 训练的好坏，而不用像之前那样盲目训练。

4.10.4 LSGAN

LSGAN 的主要目的也是采用最小二乘损失函数代替了 GAN 目标函数的交叉熵，从而解决了 GAN 训练不稳定和生成图像质量差、多样性不足的问题。

$$\begin{aligned}\min_D V_{\text{LSGAN}}(D) &= \frac{1}{2} E_{x \sim p_{\text{data}}(x)} \left[(D(x) - b)^2 \right] + \frac{1}{2} E_{z \sim p_z(z)} \left[(D(G(z)) - a)^2 \right] \\ \min_G V_{\text{LSGAN}}(G) &= \frac{1}{2} E_{z \sim p_z(z)} \left[(D(G(z)) - c)^2 \right]\end{aligned}$$

¹ Arjovsky M, Chintala S, Bottou L. Wasserstein gan[J]. arXiv preprint arXiv:1701.07875, 2017.

其中 a, b, c 属于超参数, a, b 分别表示生成图片和真实图片的标记, c 是生成器为了使判别器认为生成图片为真实样本而定的值, 这里设定 $a=0, b=c=1$ 。

论文主要回答了两个问题: 为什么最小二乘损失可以提高生成图片质量; 为什么最小二乘损失可以使得 GAN 训练更稳定。对于第一个问题, 论文认为交叉熵作为损失函数, 会使得生成器不再优化那些被判别器识别为真实图片的生成图片, 即使这些生成图片距离判别器的决策边界仍较远。原因在于生成器只需完成混淆判别器的目标即可, 而最小二乘损失则在混淆判别器的前提下还得让生成器把距离决策边界较远的生成图片拉向决策边界。对于第二个问题, 论文认为 Sigmoid 交叉熵损失容易达到饱和状态, 即梯度为 0, 而最小二乘只在一点达到饱和。

4.10.5 BEGAN

谷歌提出一种新的简单强大的 GAN, 这是一种新的评价生成器生成质量的方法, 不需要太多的训练技巧即可实现快速稳定的训练。以往的 GAN 及其变体是希望生成器生成的数据分布可以尽可能地接近真实数据分布, 因此研究者们设计了各种损失函数, 而 BEGAN 则不采用这种估计概率分布的方法, 即不直接去估计生成分布 p_g 和真实分布 p_{data} 的差距, 而是估计分布的误差分布差距, 只要分布之间的误差分布相近, 也可认为这些分布是相近的。

BEGAN 主要有 3 个贡献:

(1) 提出了一种新的简单强大的 GAN 网络结构, 使用标准的训练方式也能快速稳定地收敛。

(2) 对于生成器和判别器的平衡提出了一种均衡的概念, 提供了一个超参数, 这个超参数用于平衡图像的多样性和生成质量。

(3) 受 WGAN 启发, 提出了一种收敛程度估计。

BEGAN 采用自编码器作为判别器; 在生成器的设计上, 使用 Wasserstein 距离衍生出的损失去匹配自编码器的损失分布, 这是通过传统的 GAN 目标加上一个用来平



衡判别器和生成器的平衡项来实现的;还提出了一个衡量生成样本多样性的超参数 λ : 生成样本损失的期望与真实样本损失的期望值之比。 λ 值较低时会导致图像多样性较差, 因为此时判别器过于关注对真实图像的自编码。



第 5 章

卷积神经网络

在第 2 章中利用 MNIST 数据集训练了一个非常简单的 TensorFlow 深度学习模型，最后达到的预测准确率为 92%，但是这个准确率对于深度学习来讲并不是一个令人非常满意的结果。一个优秀的深度学习模型，可以将 MNIST 数据集的预测准确率训练到 99% 以上，那么如何提高训练的准确率呢？为了专门应对图像训练和处理的相关问题，纽约大学的 Yann LeCun 教授在 1989 年提出了卷积神经网络(Convolutional Neural Network, CNN)。

5.1 神经网络简介

5.1.1 神经元与神经网络

在深入学习卷积神经网络之前，先来了解神经网络相关的先导知识。深度学习中的神经网络概念类似于人脑神经网络，在人脑中，神经网络是由大量的神经元组成的，一个神经元通常会有很多个树突，树突是从细胞体发出的一至多个突起，呈放射状，细胞体起始部分较粗，经反复分支而变细，形如树枝，树突具有接受刺激并将冲动传入细胞体的功能，如图 5-1 所示。



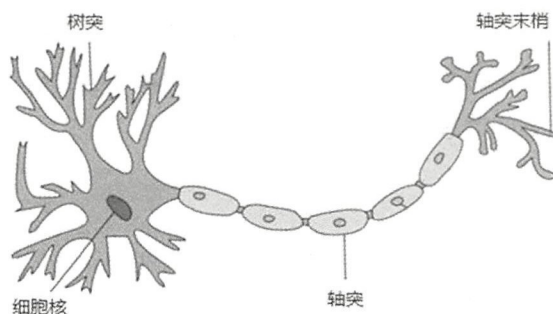


图 5-1

1943 年，心理学家 McCulloch 和数学家 Pitts 根据人的神经网络提出了神经元模型 MP，如今我们所使用的神经网络都是基于 MP 进行构建的。一个神经元模型的结构如图 5-2 所示。

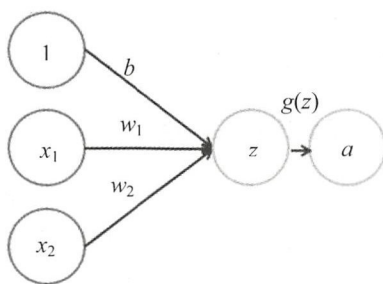


图 5-2

上面这个神经元模型与之前所讲的 $Wx+b$ 模型非常类似。在这个模型中，包含了 3 个输入和 1 个输出，用标记有字母 b , w_1 , w_2 的有箭头的线连接。在神经网络中，一般将这些线称为连接，每一根线上的字母 b , w_1 , w_2 称为权重。在神经元中，连接是一个非常重要的概念，每一个连接上都会有一个权重。使用 $g(z)$ 函数运算，最后输出。训练神经网络的过程就是通过不断调整权重，使其达到最佳，从而使整个神经网络预测的效果最好。

一般来讲，我们将数据叫作“样本”，在一个样本中，可以有很多个已知的属性，还可以有一部分未知的属性，我们会将已知的属性作为原始的数据输入一个神经元模型中，然后通过一系列的神经网络的训练，来预测出那个未知的属性是什么。我们会



用到神经网络中大量的算法进行预测，最终得到想要的值。在这个过程中，我们将已知的属性叫作特征，将未知的属性叫作目标，神经网络训练的过程，实际上就是通过已知特征预测未知目标的过程。

那么，什么是神经网络呢？

在深度学习领域中，神经网络是由多个神经元组成的，每个神经元可被描述成一个节点，每个节点实际上是一种特定的输出函数，一般称之为激励函数。每两个节点间的连接上都有一个加权值，称为权重，网络的输出则因网络的连接方式、权重和激励函数的不同而不同。因此，神经网络本身通常是对自然界某种算法或者函数的逼近，也可能是对一种逻辑策略的表达。

神经网络具有以下特点：

（1）具有自学习功能。例如，在实现图像识别的时候，我们只要提前将大量的图像样本输入神经网络，它就可以通过其自学习功能慢慢学会识别图像。

（2）具有联想存储功能。主要由神经网络的反馈网络来实现这种功能。

（3）具有快速寻找优化解的能力。在寻找一个复杂问题的优化解时，往往需要进行大量的运算，而利用神经网络的反馈特性，可以发挥计算机的高速运算能力，快速地得到优化解。

5.1.2 感知器（单层神经网络）与多层感知器

1958 年，Rosenblatt 提出了一种由两侧神经元组成的神经网络——感知器（Perceptron），它也是首个可以学习的神经网络，在当时引起了极大的轰动。感知器的模型示意图如图 5-3 所示。



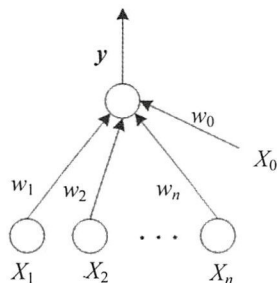


图 5-3

感知器包括输入层和输出层。其中输入层只负责传输数据，而输出层负责对前一层的输入进行运算，感知器是一个单层神经网络，在这里，我们只考虑有计算的层，而不考虑非计算的层。

之前所讲的神经元模型在进行训练的时候，其参数都是预先设定好的；而感知器模型的参数则是通过不断的训练，慢慢通过自身的学习机制逐步总结而来的，这是感知器模型与神经元模型的最大区别。

感知器最重要的思想就是定义一个算法去学习权重 w ，再将 w 乘以输入特征，以此来确定神经元是否受到了刺激。在模式分类中，可以应用这个算法确定样本是属于哪一个类别的，可使用图 5-4 来表示。

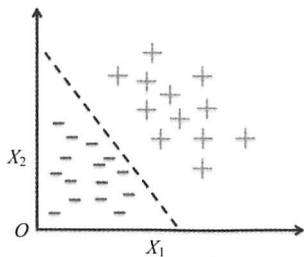


图 5-4

在深度学习领域中，感知器属于监督学习算法的一种，更具体地讲属于单层二值分类器。分类器的任务就是基于一组输入的数据，预测这个数据可能属于两个分类中的哪一个。因此，在解决二分类问题的时候，会经常使用基于感知器的模型进行学习和预测。



那么，感知器是如何学习的呢？

其实，在感知器模型中，一般会使用“阈值”来进行感知器的分类判断。简单来说，感知器在接收到多个输入信号时，会判断这些信号的总和是否超过了某个阈值，如果超过了则返回一个信号，否则就什么都不返回。因此，在使用感知器算法进行学习的时候，首先要通过自主学习得到输入信号的权值，并得出线性决策边界，进而能够区分出这两个线性可分的类是 1 还是 -1，如图 5-5 所示。

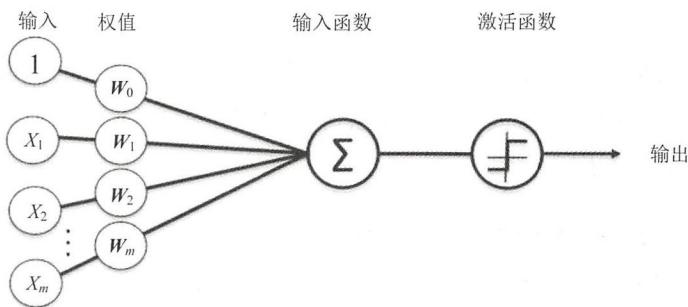


图 5-5

虽然单层感知器能够很好地解决二分类的问题，但是单层感知器却存在问题：一方面，单层感知器只能对线性可分的数据集进行分类，这很难应用在实际中，因为实际应用中的数据可能会被分为很多类，而且相对比较混乱；另一方面，单层感知器只能解决一些简单的类似于“与”“或”“非”等的逻辑运算，对于异或问题却无法解决。

为了解决这个问题，研究人员试图将多个单层的感知器进行叠加，形成一个多层感知器，从理论上讲，多层感知器具备解决任何分类问题的能力。将感知器按照一定的结构和系数进行组合，第一层感知器实现两个线性的分类器，对特征空间进行分割，而在这两个感知器输出之上再加一层感知器，就可以实现异或运算。

多层感知器 (MultiLayer Perceptron, MLP) 至少包括一个隐藏层 (除一个输入层和一个输出层外)。单层感知器只能学习线性函数，而多层感知器可以学习非线性函数。多层感知器模型如图 5-6 所示。



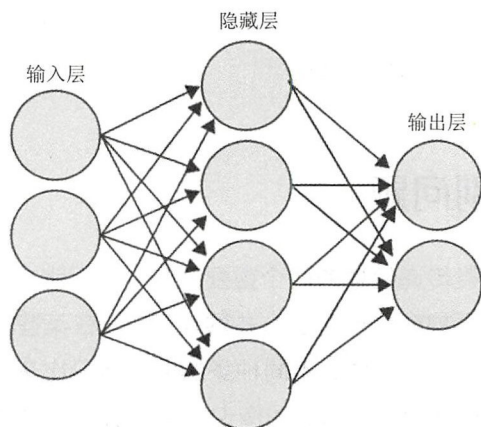


图 5-6

图 5-6 是一个典型的多层感知器模型，其中包含着一个输入层、一个输出层和一个隐藏层。在多层感知器中，所有的连接都是有权重的，也就是说，在图 5-6 中，每一个带有箭头的连接线上都有一个权重的值。

(1) 输入层：在图 5-6 中，输入层有 3 个节点，在输入层中不会进行任何计算操作，只是将输入的东西做一个输入取值。

(2) 隐藏层：在图 5-6 中，隐藏层有 4 个节点，隐藏层的节点值取决于输入层所输出的数据和输入层上每一个节点的权重，一般会在隐藏层对输入层输出的数据进行处理和相关计算，并通过一个激活函数激活，将最终获得的数据输出。

(3) 输出层：输出层有两个节点，从隐藏层接收输入，并执行隐藏层的计算。这些作为计算结果的计算值就是多层感知器的输出。

一般来讲，在深度学习领域所说的神经网络就是指多层感知器。多层感知器是由原始的感知器堆叠而成的，我们也将多层感知器称为全连接 (Full Connected) 神经网络。

在全连接神经网络中，同一层之间的神经元是没有连接的，每一层中的神经元一般只和下一层或上一层的神经元进行连接，第 N 层的每一个神经元和第 $N-1$ 层的所有神经元相连接，这时，第 $N-1$ 层的神经元的输出就是第 N 层神经元的输入。在全连



接神经网络中，每一个连接都有一个确定的权重，一般在卷积神经网络中的最后一层或两层都是由全连接神经网络组成的。

5.2 图像识别问题

文字和图像是人类传递信息的两个重要的方式，相比文字而言，图像能够更加生动形象地以一种人类更加容易理解的方式展现信息，在某些情况下甚至更具有艺术气息。人之所以能够将不同的物体、不同种类的图像区分出来，是因为每一种图像都具有独一无二的特征，在计算机领域，图像识别是计算机视觉中非常重要且非常基本的内容，也是图像检测、分割、物体跟踪等高层次视觉任务的基础。在现实生活中，人们经常利用图像识别技术完成一些生活中常见的问题，例如，公安系统利用人脸识别技术进行犯罪嫌疑人的照片对比，物流公司利用智能分析技术来判断司机在驾驶的过程中是否有违规行为等。

如图 5-7 所示，当人们看到这张图的时候，脑海中第一反应就是：这是一只猫，并且能够通过颜色判断出它是一只灰色的猫。那么为什么我们一下子就能够认出来这是一只猫呢？其原因就是在看到这只猫之前，我们的脑中已经存储了大量猫的图像，我们看到过各种各样的猫，并且通过不同样子的猫，学习到了猫相似的特征，当给出图 5-7 时，人的潜意识中就会去主动地提取图 5-7 的特征，再将脑海中所看过的各种各样的事物的特征加以比对，最后发现，这个特征和猫的特征最接近，于是就可以判断出这是一只猫。

假设我们给这幅图像设定四个可能的标签，分别是猫、狗、帽子、杯子，对于计算机而言，图像是一个非常大的三维数组。在图 5-7 中，猫的图片是由 248 像素宽和 400 像素高组成的，并且有 RGB 三个通道，也就是说，这里面有 $248 \times 400 \times 3 = 297\ 600$ 像素，并且每像素的范围都是在 0~255 之间的整数。对于计算机而言，就是将这些像素点通过各种组合形成一个标签——猫。



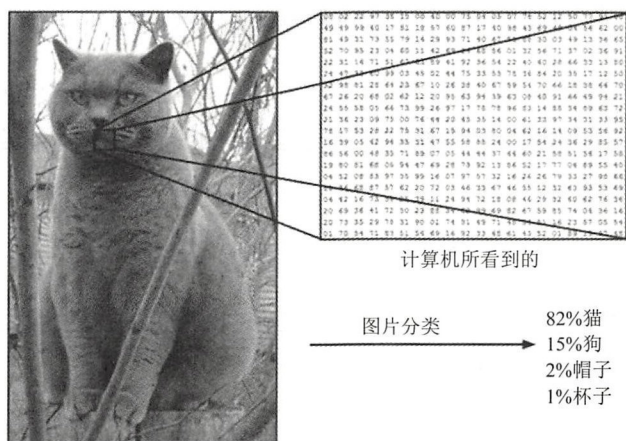


图 5-7

对于人来说，识别一幅猫的图像易如反掌，但是对于计算机来说，这却是一个复杂而又艰巨的任务。即使是同一个事物，由于外界环境的不同，对计算机来讲也可能就是完全不同的两个事物。一般来讲，影响计算机判断的因素包含且不仅限于以下几个方面。

(1) 视角的变换：即使是同一个事物，但是由于观察角度的不同，所呈现出来的样子也完全不同。

(2) 尺度的变换：有一些事物的大小不是固定的，例如同一个气球，在充满气的时候和没有充满气的时候所呈现出来的大小是完全不同的，甚至当气球中的气压严重不足时，在外形方面也会有非常大的变化。

(3) 物体的扭曲：在现实生活中有很多物体由于外界环境的因素导致其本身的形状发生了改变，例如物体受到挤压导致变形，甚至极度的扭曲。

(4) 遮挡：这在图像处理中是非常常见的一种现象，例如在识别照片中的某个人物的时候，由于拍照的时候目标人物可能会被遮挡，导致我们仅能够根据其现有的信息提取少量的特征，甚至有时只有几个像素可见。

(5) 光照条件：任何一个物体在不同的光照环境下所呈现出来的状态都是不一样的。



(6) 背景干扰：目标物体可能跟其他物体相互混杂，使其很难被分辨和发现。

以上这些问题对计算机进行目标识别和分类都有极大的影响，而我们的目的就是通过学习深度学习相关技术教会计算机如何在这些恶劣的环境因素下精准地识别出物体，并将物体准确地分类。

那么怎样使用深度学习技术来解决图像识别的问题呢？

其实对于机器来说，无论是多少个分类的图像识别问题，其流程都是一致的，总体归纳起来大致可以分为以下几个步骤。

(1) 输入：一般会给定 K 个类别的 N 幅图像，作为计算机学习的训练集，让其根据训练集进行学习。

(2) 学习：将训练集输入计算机，让计算机去观察每一幅图像，然后将所观察到的图像进行归纳和总结。

(3) 评估：当计算机完成学习这个步骤的时候，就需要给一幅它之前没有学习过的图像，让它自己去判断这幅图像到底属于哪个分类，再将其得到的结果与已知的结果进行对比。

(4) 调参：往往经过评估后，我们会发现，也许在设定这个模型的时候，有很多情况没有考虑周全，就像编写代码运行之后产生的运行时异常一样。对于这种暴露出来的问题，我们需要根据实际情况去调整相应的参数，使其离优秀的模型更进一步。

(5) 生成：重复步骤(3)和步骤(4)，就会得到一个看上去最优的模型，将生成的这个最优的模型作为生产模型，帮助我们去解决实际的问题。

虽然使用深度学习技术能够解决很多图像识别的问题，但是在实际使用过程中还是存在着很多难点。

比如针对行业划分到很细粒度的图像的识别，一直就是图像识别的难题，其根本难点在于：由于细粒度图像属于某个垂直领域，所以造成样本量不够的问题，并且没有经过专业训练的人是难以辨认的。对于机器而言，在细粒度图像的样本量不大或清晰度不高的情况下，直接通过深度学习进行训练而得出的分类器基本上都是欠拟合的，



很难学习到其局部的细微特征；而且，细粒度图像在识别的时候，最关注的就是其局部特征，即使样本图像是非常清晰的，由于其局部特征非常多，机器也很难从众多的局部特征中获取最为有用的局部特征。这对于深度学习而言是一个很大的难题，即使对于人类而言，如果不是相关领域的专家，仅凭肉眼也很难正确地找到区分特征。

5.3 常用的图像库介绍

对于机器学习和深度学习来说，最重要的就是拥有大量且合适的数据集，可用于深度学习模型的训练和检验。使用优秀的数据集不仅有利于加快训练速度，提高训练精度，而且还可以使程序开发人员避免在模型建立过程中犯的一些算法错误，改善程序运行的结果和提高执行效率。对于一般的开发公司及个人开发者来说，要想自己建立一套数据集，工作量是无比庞大的。为了免去程序开发人员自己建立数据集的精力耗费，有组织和公司免费提供了一些用于训练和测试模型的图像数据库。

下面就来介绍一下常用的图像集数据库，以及这些数据库的用法和用处。

1. MNIST

MNIST 数据集在第 2 章曾经用过，也做了相关介绍，是图像分类经常用到的一个数据集，也是深度学习和神经网络的初学者第一个碰到的数据集。在这个数据集中，包含了几万张由 28 像素×28 像素组成的数字 0~9 的手写数字，并且只包含了图像的灰度值信息。在这个数据集中包含 55 000 张训练集、10 000 张测试集，以及 5 000 张验证集，如图 5-8 所示。

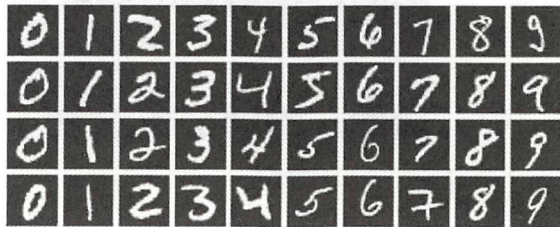


图 5-8

在 TensorFlow 中下载与读取 MNIST 数据集的方法如下:

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("MNIST_data/", one_hot = True)
```

2. CIFAR-10

对于手写体和数字识别来说, MNIST 是一个非常有用的数据集, 但是对于各种物体及更高级一些的图像分类应用来说, MNIST 数据集就显得逊色了很多, 使用 CIFAR-10 数据集则更为合适。

CIFAR-10 数据集包含了 60 000 张 32 像素×32 像素的彩色图片, 其中包含 50 000 张图片用来进行模型训练, 有 10 000 张图片用来进行模型的验证。在 CIFAR-10 数据集中, 将图片分成了 10 类, 每一类有 6 000 张图片, 这 10 类分别是 airplane (飞机)、automobile (汽车)、bird (鸟)、cat (猫)、deer (鹿)、dog (狗)、frog (青蛙)、horse (马)、ship (轮船)、truck (货车), 如图 5-9 所示。

在 TensorFlow 中直接使用 import cifar10, cifar10_input 来导入数据集。



图 5-9

另外, 如果 CIFAR-10 数据集所包含的分类还是无法满足需求, 那么可以使用 CIFAR-100 数据集, 它提供了 100 类、20 个超类的图片供程序开发人员来使用。

3. STL-10

如果开发者感觉使用 CIFAR-10 数据集时的图像较小且清晰度不够的话, 还有 STL-10 数据集供开发者选择。STL-10 数据集也是一个图像数据集, 与 CIFAR-10 相似, 同样包含了 10 类物体的图像, 每类含有 13 000 张图片、500 张训练集和 800 张测试集, 分辨率为 96 像素 \times 96 像素, 是 CIFAR-10 数据集图像大小的两倍。另外, 除了具有类别标签的图片之外, 还有 100 000 张无类别信息的图片供开发者训练和测试用, 如图 5-10 所示。



图 5-10

4. ImageNet 数据集

ImageNet 是计算机视觉领域中的一个非常常用的数据集, 也是目前世界上最大的图像识别数据库, 是美国斯坦福大学的科学家模拟人类的识别系统建立的, 主要的用途是从图像中识别物体。ImageNet 是一个非常有前景的研究项目。

ImageNet 数据集是按照 WordNet 架构组织的大规模带标签图片的数据集。在这

个数据集中，大概有 1 500 万张图片，2.2 万个分类，每张图片都经过了严格的人工筛选和标记。

ImageNet 就像一个网络一样，拥有许多个节点，每个节点相当于一个 item 或者 subcategory。据官网得知，一个节点目前至少含有 500 个对应物体的可供训练的图片。它实际上就是一个巨大的科工图像/视觉训练的图像库。ImageNet 的结构基本上是金字塔形的：目录→子目录→图片集。

5. KTH-ANIMALS

如果需要大量的动物图像集，那么 KTH-ANIMALS 数据集就是一个非常好的选择。它提供了 19 种不同类别的图像，每一个类别中大约有 100 张不同大小的图片，并且也提供了前景和背景的信息，如图 5-11 所示。

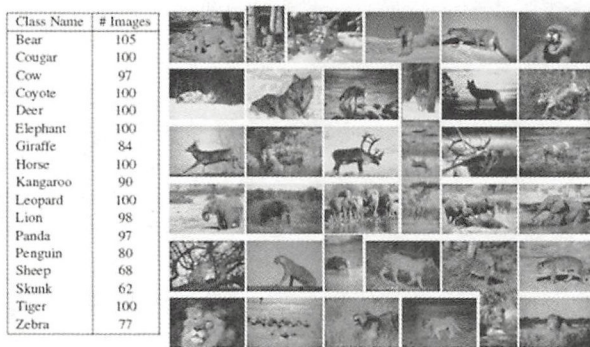


图 5-11

5.4 卷积神经网络简介

卷积神经网络（Convolutional Neural Network，CNN）是深度学习技术中常见的神经网络结构之一，在图像识别处理和语音分析领域中起着无法替代的作用。相比其他的网络处理模型而言，利用卷积神经网络进行图像和语音分析，可以大大地降低网络模型的复杂度，减少了权值共享的数量，使训练速度有大幅度的提升，图像的维度越高，其优越性体现得越明显。

卷积神经网络 (CNN) 从结构上看本质是多层感知器, 而相对于其他多层感知器而言, 卷积神经网络的多层感知器是为了识别二维形状而特别设计的。从第 2 章中可以知道手写体的图像实际上是一个 28×28 的二维向量, 因此, 使用 CNN 进行手写体训练是非常适合的。

全连接神经网络可以很容易地解决分类问题, 但是解决图像识别问题会变得异常麻烦。因为图像是由二维像素点组成的, 如果使用全连接神经网络处理一幅 $2\,000$ 像素 \times $2\,000$ 像素的图像, 输入层就会有 $2\,000 \times 2\,000 = 4\,000\,000$ 个节点, 假定隐藏层只有 200 个节点, 那么仅仅是这一层就会有 $(2\,000 \times 2\,000 + 1) \times 200 = 8$ 亿个参数, 这个数量是相当惊人的, 消耗的资源会非常大, 速度也会变得非常慢, 而且其可扩展性能也非常差, 这并不是我们想要的。

再加上, 使用全连接神经网络进行处理的数据层数最多不超过 3 层, 因此一般先降维, 使图像样本符合一个全连接神经网络能够处理的要求。这样做就不可能得到一个深度很大的全连接神经网络, 大大地限制其能力。

正是因为全连接神经网络在进行图像处理方面有如此多的限制, 所以在 20 世纪 60 年代, Hubel 和 Wiesel 提出了卷积神经网络, 专门解决图像识别问题。

5.4.1 CNN 的基本原理与卷积核

首先从一个非常经典的卷积神经网络与全连接神经网络的对比来说明二者的区别, 如图 5-12 所示。

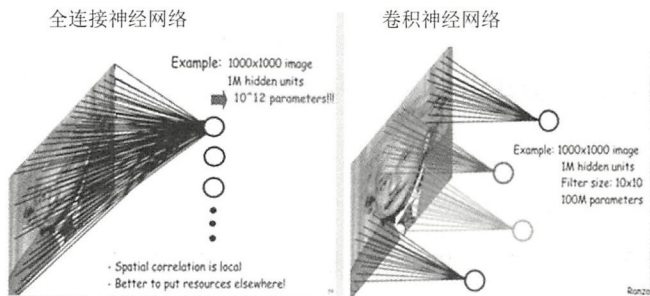


图 5-12

在图 5-12 中，左侧是一个经典的全连接神经网络，右侧是卷积神经网络。

图 5-12 左侧样本是由 $1\,000$ 像素 \times $1\,000$ 像素组成的一幅图像，在输入层后，如果设定下一个隐藏层的神经元数量为 10^6 个，那么采用全连接神经网络就会有 $1\,000 \times 1\,000 \times 10^6 = 10^{12}$ 个权值参数，这么多的参数对于计算机来讲训练起来是非常困难的；但是如果采用局部连接的方式，先设定一个固定大小的区域进行连接对比，在图 5-12 右图中，使用 10×10 的大小作为与其对比的单位，此时的权值参数的数量则为 $10 \times 10 \times 10^6 = 10^8$ 个，与之前的 10^{12} 相比而言，权值个数减少了 4 个数量级。

一般将这个 10×10 大小的单位称为卷积核。卷积核实际上就是一个权值矩阵，表示如何处理单像素和与之相邻的邻域像素之间的关系。一般来说，卷积核的尺寸跟计算量成正比关系，当卷积核尺寸较大时，其计算量也会成倍增加。目前来说，应用最多的为 3×3 大小的卷积核。在某些特定的情况下，会选取大小为 1×1 的矩阵作为卷积核，卷积核可在多通道的神经网络中实现跨通道之间的交互和信息整合。同时卷积核可使通道数很方便地降维和升维。

通过卷积核的形式，我们将 10^{12} 个权值成功地减少到了 10^8 个，但是能不能更进一步地简化权值数量呢？答案显然是可以的，即采用权值共享的方法，将卷积核的权值共享给剩下的部分，使其能够被重复利用，以降低权值的数量，减少计算量，那么具体怎么做呢？

在上面的例子中，设定了隐藏层的每个神经元都连接了 1 个 10×10 的卷积核，用这个卷积核去扫描这幅目标图像，这时，目标图像中的每一个位置都是被一个同样的卷积核进行扫描，而每个卷积核中都包含着特定的权值（也就是我们说的权重信息），所以图像中每个被扫描到的位置的权重实际上是一样的，即权值共享。权值共享的存在，使得计算量会变得相对较小，有利于卷积神经网络的训练。

5.4.2 池化

在卷积神经网络中，池化（Pooling）也是一个非常重要的概念，其核心目的就是采样。在深度学习领域中，有很多种池化函数，如 `max_pool()` 函数、`avg_pool()` 函数等。这两个函数也是 TensorFlow 中最常见的池化函数。以 `max_pool()` 函数为例，它将

输入的图像划分为若干个子区域，并输出每个子区域的最大值。这种机制有效的原因在于，在发现一个特征之后，它的精确位置远不及它与其他特征的相对位置的关系重要。池化层会不断地减小数据的空间大小，因此参数的数量和计算量也会下降，这在一定程度上也控制了过拟合。通常来说，CNN 在卷积层之间都会周期性地插入池化层。

池化层对输入的特征图（Feature Map）都会选择一种方式进行压缩，这也可以理解为降采样，即对采样的特征图进行降维。对于 CNN 来说，降维的作用在于减少了输入参数，降低了计算的复杂度，提高计算效率，对后续层中所需要的参数也有了大幅度的缩减。avg_pool()函数就是把很多数据用其平均值代替，以降低数据的复杂度。

但不管是采用什么样的池化函数，当我们对输入的改变非常少的时候（例如非常少量的平移操作），池化函数最后的输出结果一般不会发生改变或仅仅发生极少数的改变，对于 CNN 来说，这叫作“平移不变性（Translation Invariant）”。平移不变性对于神经网络的训练是有益的，正是因为平移不变性，让我们通过池化层将周围的特征进行整合时，只需关心采样的特征出现与否，而不必关心其出现的位置。

在 TensorFlow 中，一般会使用以下两种函数进行池化操作：

```
tf.nn.max_pool(value, ksize, strides, padding, data_format='NHWC', name=None)
tf.nn.avg_pool(value, ksize, strides, padding, data_format='NHWC', name=None)
```

可以看到，在 TensorFlow 中池化函数的输入实际上是相同的，解析以下这些输入参数：

- **value**: 一个 4 维的张量，一般是卷积后的特征图（Feature Map），形状为[batch, height, width, channels]；
- **ksize**: 池化窗口的大小，一般是一个长度不小于 4 的数组；
- **strides**: 输入张量的每个维度的滑动窗口的步幅，一般是一个长度不小于 4 的数组；
- **padding**: 一个字符串，取值为 SAME 或者 VALID；
- **data_format**: 'NHWC'代表输入张量维度的顺序，N 为个数，H 为高度，W 为宽度，C 为通道数；
- **name**: 自定义操作的名称。

5.4.3 再探 ReLU

前面简单地介绍 ReLU 这种激活函数，知道 ReLU 函数相对于其他函数而言具有收敛速度快的特点，并且，只需要一个阈值就可以得到激活值，而不需要进行一大堆的复杂运算，其公式为

$$f(x) = \max(0, x)$$

其实，CNN 往往在选择激活网络的时候，更加倾向于 ReLU 函数，这是因为它可以将神经网络的训练速度提升数倍，而并不会显著地影响模型的泛化准确率。

回忆一下计算梯度的公式 $\nabla = \sigma' \delta x$ ，其中 σ' 是 Sigmoid 函数的导数。在使用反向传播算法进行梯度计算时，每经过一层 Sigmoid 神经元，梯度就要乘以一个 σ' 。如图 5-13 所示。从图 5-13 中可以看出，如果使用 Sigmoid 函数来求解梯度下降的问题， σ' 函数的最大值应该为 0.25，如果是这样的话，当乘以一个 σ' 则会导致梯度变得越来越小，即会产生梯度消失，这对于深层网络训练是一个很大的问题。ReLU 函数的出现从很大程度上解决了梯度消失的问题，而且真正使得深度学习的工程实用性大大提升，可以更简单地加深网络层数，达到较好的效果。

那么，ReLU 函数具体是如何工作的呢？实际上，ReLU 函数通过滤波器以像素为单位进行卷积扫描的时候，利用其本身的特性，将所有值为负数的像素都用 0 来替换，其目的是向卷积神经网络中引入非线性的相关操作，因为在日常生活中我们大多数所见到的及需要学习的知识都是非线性的。

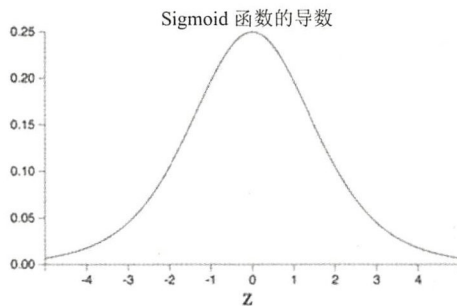


图 5-13

到这里，再来介绍一下 ReLU 函数的另外一个特性——稀疏激活性。从信号角度看，神经元只对少部分输入信号选择性响应，更多信号被刻意地屏蔽了，这样可以提高学习的精度，更好更快地提取稀疏特征。因此，如果使用传统的 Sigmoid 函数进行计算和训练的话，就会同时激活接近一半的神经元，但是从神经科学的研究角度来看，这是不可行的。从另一方面来讲，如此大量的神经元被激活，势必会导致计算量剧增，给神经网络训练带来巨大的负担。

在进行深度学习计算的时候，ReLU 函数会使一部分神经元 (x 负轴) 的输出为 0。也就是说对于一个神经网络而言，使用 ReLU 函数就表明 x 负轴没有被激活。这样使其参数变得很少，大大地减少了神经网络的运算量，而且还会使大量不相关的信号被屏蔽掉，更有效率地提取重要的特征。

5.5 CNN 模型

在前面的章节中，我们详细地介绍了 CNN 在深度学习和神经网络中扮演的角色及作用。接下来，将介绍一些经典的 CNN 模型及其对比。

5.5.1 LeNet-5 模型

LeNet-5 模型是由纽约大学的 Yan LenCun 提出的经典卷积神经网络模型，基本上所有讲 CNN 的资料都会对该模型进行简单的介绍。该模型主要用来识别手写体，曾被广泛应用于美国银行支票的手写体识别。该模型的网络结构如图 5-14 所示。

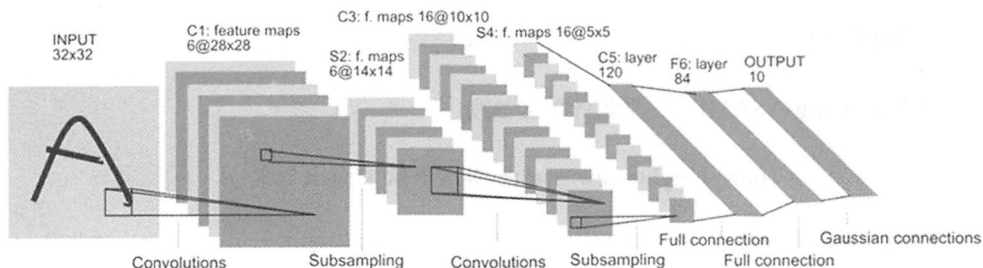


图 5-14

由图 5-14 可见，LeNet-5 模型由 8 层组成：输入层（INPUT）、C1 层、S2 层、C3 层、S4 层、C5 层、F6 层、输出层（OUTPUT）。下面，我们来详细地介绍这 8 层在 LeNet-5 模型中的作用。

输入层（INPUT）：LeNet 模型的输入层是一个 32 像素×32 像素的单通道图像，即灰度图，比 MNIST 数据集中最大的字母还要大，便于潜在的明显特征能够出现在最高层特征监测感受野的中心。

C1 层：C1 层为卷积层，一共包含 6 个特征图。因为卷积核的尺寸为 5×5，步长为 1，padding 的方式为 VALID，所以特征图像的尺寸为 28 像素×28 像素。那么这个尺寸是怎么得来的呢？根据 TensorFlow 中的 conv2d 函数，我们首先定义几个基本符号：

（1）输入矩阵 $W \times W$ ，在这里一般只考虑宽和高相等的情况，因为如果宽和高不相等，那么推导方式也会不同。

（2）Filter 矩阵 $F \times F$ ，这个值为卷积核的大小，在例子中，其大小为 5×5。

（3）Stride 值 S ，也就是步长。

最终输出的宽和高分别为 new_height、new_width。在 TensorFlow 中，padding 的方式有两种：VALID 和 SAME。如果 padding="VALID"，那么计算公式为

$$\text{new_height}=\text{new_width}=(W-F+1)/S \text{（结果向上取整）}$$

也就是说，conv2d 的 VALID 方式不会在原有输入的基础上添加新的像素，在这里假定输入的是图像数据，因为只有图像数据才会有像素这个单位，而此时输出的矩阵大小直接按照公式计算即可。

如果 padding="SAME"，那么计算公式为

$$\text{new_height}=\text{new_width}=W/S \text{（结果向上取整）}$$

因此，根据上述的公式，输入的矩阵为 32×32，卷积核的大小为 5×5，步长为 1，高度和宽度均为 $(32-5+1)/1=28$ ，最后得到的结果为 28 像素×28 像素的特征图尺寸。

S2 层为下采样层,也叫池化层,每个单元与 C1 所对应的特征图上 2 像素×2 像素的区域连接,这四个输入相加后乘以一个可训练的系数,再加上一个可训练的偏置量后应用 Sigmoid 函数,2 像素×2 像素的感受野不重叠,所以 S2 的特征图尺寸为 14 像素×14 像素。

C3 层为卷积层,一共包含 16 个特征图。因为卷积核的尺寸为 5×5,步长为 1, padding 的方式为 VALID,据公式可以算出,其最后的宽和高均为(14-5+1)/1=10 像素,最后得到的结果为 10 像素×10 像素的特征图尺寸。C3 中每个单元连接到 S2 特征图子集的 5 像素×5 像素区域,具体的连接方式如图 5-15 所示。

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

图 5-15

S4 层为池化层,每个单元与 C3 所对应的特征图上的 2×2 区域连接,这四个输入相加后乘以一个可训练的系数,然后加上一个可训练的偏置量后应用 Sigmoid 函数进行激活,此时 2×2 的感受野不重叠,所以 S2 的特征图尺寸为 14 像素×14 像素。

C5 层为卷积层,一共包含 120 个特征图,每个单元连接到 S4 所有特征图上的一个 5×5 的区域。因为卷积核的尺寸为 5×5,步长为 1, padding 的方式为 VALID,据其公式可以算出,最后的宽和高均为(5-5+1)/1=1,得到的结果为 1×1 的特征图尺寸。

F6 层为全连接层,共包含 84 个神经元。将该层的输入与权重做点乘积,然后加上偏置量后应用 Sigmoid 函数进行压缩。

输出层 (OUTPUT) 包含了 10 个神经元,分别对应于 10 种分类的结果。该层由欧几里得径向基函数 (RBF) 组成,每个 RBF 单元的输出由以下公式得出:

$$y_i = \sum_j (x_j - w_{ij})^2$$

在上面的过程中，主要用到了卷积和下采样（池化），接下来简单地介绍它们的主要过程。

（1）卷积过程：用一个可训练的滤波器 f_x 去卷积输入的图像，这个输入的图像在第一个阶段是图像，到了后面的阶段实际上就是卷积特征映射，然后将这个输入的图像加上一个偏置量 b_x ，得到卷积层 C_x 。在卷积运算中，最重要的一个特点就是通过卷积使原信号的特征增强，并且降低噪声。

（2）下采样过程：将每个相邻区域的四个像素求和变为一个像素，然后通过标量 W_{x+1} 进行加权，再增加偏置量 b_{x+1} ，通过 Sigmoid 激活函数产生一个缩小 1/4 的特征映射图 S_{x+1} 。

下面使用 TensorFlow 来实现一个 LeNet 模型：

```
def inference(inputs):
    # input shape: [batch, height, width, 1]
    with tf.variable_scope('conv1'):
        weights = tf.Variable(tf.truncated_normal([5, 5, 1, 6], stddev=0.1))
        biases = tf.Variable(tf.zeros([6]))
        conv1 = tf.nn.conv2d(inputs, weights, strides=[1, 1, 1, 1], padding='VALID')
        conv1 = tf.nn.relu(tf.nn.bias_add(conv1, biases))
    maxpool2 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1])
    with tf.variable_scope('conv3'):
        weights = tf.Variable(tf.truncated_normal([5, 5, 6, 16], stddev=0.1))
        biases = tf.Variable(tf.zeros([16]))
        conv3 = tf.nn.conv2d(maxpool2, weights, strides=[1, 1, 1, 1])
        conv3 = tf.nn.relu(tf.nn.bias_add(conv3, biases))
    maxpool4 = tf.nn.max_pool(conv3, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1])
    with tf.variable_scope('conv5'):
        weights = tf.Variable(tf.truncated_normal([5, 5, 6, 16], stddev=0.1))
        biases = tf.Variable(tf.zeros([16]))
        conv5 = tf.nn.conv2d(maxpool4, weights, strides=[1, 1, 1, 1])
        conv5 = tf.nn.relu(tf.nn.bias_add(conv5, biases))
    with tf.variable_scope('fc6'):
        flat = tf.reshape(conv5, [-1, 120])
        weights = tf.Variable(tf.truncated_normal([120, 84], stddev=0.1))
        biases = tf.Variable(tf.zeros([84]))
        fc6 = tf.nn.matmul(flat, weights) + biases
        fc6 = tf.nn.relu(fc6)
```

```
with tf.variable_scope('fc7'):  
    weights = tf.Variable(tf.truncated_normal([84 10], stddev=0.1))  
    biases = tf.Variable(tf.zeros([10]))  
    fc7 = tf.nn.matmul(fc6, weights) + biases  
    fc7 = tf.nn.softmax(fc7)  
return fc7
```

5.5.2 AlexNet 模型

5.5.1 节重点讲解了 LeNet-5 模型,并用 TensorFlow 框架对其进行了代码的实现,我们知道,LeNet-5 模型主要用于识别 10 个数字的手写体,虽然对其稍加改造就能够在 ImageNet 数据集上进行应用,但是效果非常差,如果想在类似 ImageNet 的数据集中进行更好的训练,AlexNet 模型绝对是一个非常好的选择。

AlexNet 模型诞生于 2012 年,当时其作者参加 ImageNet 比赛获得冠军,由于这个模型的作者名字是 Alex,所以后来人们将这个模型命名为 AlexNet 模型。AlexNet 模型对于 CNN 具有重要的意义:AlexNet 模型直接证明了 CNN 在复杂模型下的有效性,GPU 又促使训练在可接受的时间内得到结果,推动了监督学习的发展。AlexNet 模型的网络结构如图 5-16 所示。

由图 5-16 可见,AlexNet 模型是由 8 层组成的,在这 8 层中,前 5 层为卷积层,后面 3 层为全连接层,下面,我们来详细地介绍一下这 8 层在 AlexNet 模型中的作用。

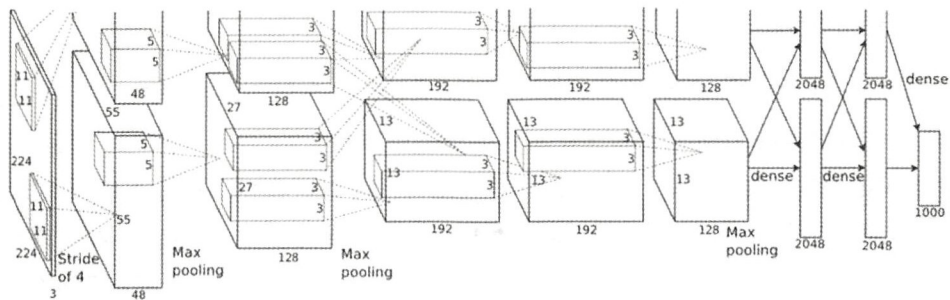


图 5-16

可以回想一下在 5.5.1 节中学习的 LeNet-5 模型,那个模型在处理卷积的时候用的是相对传统的方式,但是从图 5-16 中可以看出,AlexNet 模型的卷积部分是由上下

两部分组成的，也就是说，在卷积层，我们会把最后计算结果分开处理。

第一层：AlexNet 模型的第一层是卷积层，该层输入为 227 像素×227 像素的图像（在官方公布的论文和很多资料中，都将其输入的大小看作 224 像素×224 像素，其实这个结论是错误的），采用的是 3 通道的图像，也就是常说的 RGB 图像。该图像使用了 96 个卷积核，每个卷积核的大小为 11×11，一般也会将其称为一个 11×11 的滤波器，并用它来进行特征的提取。在进行卷积的时候，使用公式 $(\text{image_size} - \text{filter_size}) / \text{stride} + 1$ 来计算特征的大小，代入公式得到 $(227 - 11 + 2 \times 0) / 4 + 1 = 55$ ，因此，所得新的特征图的大小为 55×55，由于 AlexNet 模型在处理卷积的时候，是分上下两层来处理的，并且在第一层中有 48 个卷积核，第一层卷积后则得到了 $55 \times 55 \times 48 \times 2 = 290\,400$ 像素大小的特征图，并且这个特征图是一个彩色的且有 RGB 通道的。

在得到第一层的特征图后，就可以先使用 ReLU 函数进行激活了，在这里之所以使用 ReLU 函数，是因为相比 Softmax 函数而言计算量更少，收敛更快，使得特征值的范围更加合理，再使用池化函数来降采样处理。在这里，池化层的内核大小为 3×3，通过前面的章节，我们可以了解到，池化的过程实际上就是对这个 3×3 大小的区域进行降采样处理，最后得到一个边为 $[(55 - 3) / 2 + 1] = 27$ 的特征图，再将这些特征图作为输入矩阵，进行第二次卷积。

第二层：通过图 5-16 可以看到，在第二层卷积处理的过程中，使用了 256 个 5×5 大小的卷积核，并使用这个卷积核对输入的 27×27×96 个特征图进行卷积计算，进一步提取相关的特征。

第二层卷积和第一层卷积在处理方式上稍有不同，在第二层卷积中，滤波器是对这 96 个特征图中的一部分特征图中相应的区域乘以相应的权重，再加上一定的偏置量之后所得到的区域进行卷积，而并非对全部特征图进行卷积。在进行卷积的时候，同样是使用公式 $(\text{image_size} - \text{filter_size}) / \text{stride} + 1$ 来计算特征大小的，代入公式得到 $(27 - 5 + 2 \times 2) / 1 + 1 = 27$ ，因此，我们所得新的特征图的大小为 27×27，个数为 256 个，再对其进行 ReLU 操作，以及使用池化函数进行降采样处理，最后得到一个边为 $[(27 - 3) / 2 + 1] = 13$ 的特征图，其特征图的大小为 13×13，个数为 256 个。最后再将这些特征图作为输入矩阵，进行第三次卷积。

第三层：在第三层中，我们同样使用上面的方法进行计算，可以得出，新的特征图的边为 $(13-3+2\times 1)/1+1=13$ ，其大小为 13×13 ，个数为 384 个，所以在本层中，新的特征图个数为 $13\times 13\times 384=64\ 896$ 个。值得注意的是，在第三层的计算过程中，与前两层相比最大的区别就是，第三层没有使用池化层进行降采样。

第四层：在第四层中的处理方式与第三层是类似的，实际上，第四层的处理过程仅仅是对第三层的结果做了一个 ReLU 激活操作而已。最后同样获得了新的特征图个数为 $13\times 13\times 384=64\ 896$ 个。在第四层中，同样没有使用池化层进行降采样。

第五层：第五层的处理方式实际上与第四层的处理方式是类似的，也是对上一层（第四层）进行了一个 ReLU 操作后生成的，不同的就是其卷积的个数为 256 个，得到了一个 $13\times 13\times 256$ 大小的特征图，并且最后进行了一个池化操作，代入公式，得到一个边为 $[(13-3)/2+1]=6$ 的特征图，其特征图的大小为 6×6 ，个数为 256 个。值得注意的是，在这里使用的池化操作为最大池化操作。至此，在 AlexNet 模型中的卷积操作已经全部完成，接下来从第六层开始就会使用五层卷积得到的结果进行全连接处理。

第六层：第六层是 AlexNet 网络中的第一个全连接层，实际上是在第五层池化后进行的全连接处理。在对第五层进行全连接后，得到的特征图的大小由 13×13 降为 6×6 ，并对 256 个特征图进行全连接后，得到了 4 096 个节点，这也是在本层中最后所输出的节点个数。

值得注意的是，在这一层的最后有一个特殊的操作层——dropout 层，dropout 层的操作方式是从 4 096 个节点中随机丢弃一些节点，也就是说，随机地将一些节点信息进行了值清零操作。

第七层：第七层的操作方式实际上和第六层完全一样，最终也得到了新的 4 096 个节点。

第八层：第八层是 AlexNet 网络中的第三个全连接层，也是整个 AlexNet 网络中的最后一层。在这一层中，采用的是 1 000 个神经元，然后对第七层中的 4 096 个节点进行全连接运算，利用 Softmax 函数最终得到 1 000 个浮点型的值，这个值一般将

其描述成 1 000 个类的概率值。

下面使用 TensorFlow 来实现一个 AlexNet 模型：

```
import tensorflow as tf
slim=tf.contrib.slim
import os
import time

trunc_normal=lambda stddev: tf.truncated_normal_initializer(0.0,stddev)
height,width=299,299

def conv2d(input_, output_dim, k_h=3, k_w=3, d_h=1,
d_w=1,padding='SAME',stddev=1e-1,name="conv2d"):
    #定义一个通用的卷积层函数
    with tf.variable_scope(name):
        w=tf.get_variable('w', [k_h,k_w,input_.get_shape()[-1],output_dim],
initializer=tf.contrib.layers.xavier_initializer())
        conv = tf.nn.conv2d(input_, w, strides=[1, d_h, d_w, 1], padding=padding)

        biases = tf.get_variable('biases', [output_dim], initializer=tf.constant_
initializer(0.0))
        conv = tf.nn.bias_add(conv, biases)

        conv=tf.nn.relu(conv)
        return conv
def linear(input_, output_size, stddev=1e-1,
bias_start=0.0,relu=True,name='Linear'):
    #定义一个通用的全连接层
    shape = input_.get_shape().as_list()
    with tf.variable_scope(name):
        w = tf.get_variable("w", [shape[1], output_size], tf.float32,
tf.random_normal_initializer(stddev=stddev))
        bias = tf.get_variable("bias", [output_size],
initializer=tf.constant_initializer(bias_start))
        fc=tf.nn.xw_plus_b(input_, w, bias, name='fc')
        if relu:
            return tf.nn.relu(fc)
        else:
            return fc
```

```
class AlexNet(object):
    def __init__(self, sess, batch_size=64, dataset_name='default', checkpoint_dir=None):
        self.y_dim=1000
        self.batch_size=batch_size
        self.dataset_name=dataset_name
        self.checkpoint_dir=checkpoint_dir
        self.frame_size=3
        self.build_model()
        self.sess=sess

    def build_model(self):
        #构建模型
        image_dims=[height,width,self.frame_size]
        y_dim=[self.y_dim]

        #定义网络入口
        self.inputs=tf.placeholder(tf.float32,[self.batch_size]+image_dims,
name='real_images')
        self.y=tf.placeholder(tf.float32,[self.batch_size]+y_dim,name='y')

        #定义 forward 和 loss
        self.y_logit=self.alexmodel(self.inputs)
        self.loss=tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logist
(labels=self.y,logist=self.y_logit))

        #定义模型保存
        self.saver=tf.train.Saver()
    def train(self,config):
        #定义优化器
        en_optim=tf.train.AdamOptimizer(config.learning_rate,betal=config.
betal).minimize(self.loss)
        try:
            tf.global_variables_initializer().run()
        except:
            tf.initialize_all_variables().run()

        start_time=time.time()

        #定义计数器
        counter=0
```



```

#开始迭代训练
print('start training...')
for ae_epoch in range(config.epoch):
    batch_idx=config.train_size

    for i in range(batch_idx):
        #定义输入, 这里以随机数代替, 读者根据自身需求更改输入
        inputs=tf.random_uniform((self.batch_size,height,width,3))
        labels=tf.one_hot(tf.constant(1,shape=[self.batch_size]),
depth=1000,on_value=1.0,off_value=0.0)
        #这里需要先从定义的 inputs 和 labels 等张量得到具体的数字
        batch_inputs,batch_labels=sess.run([inputs,labels])

        _,loss=self.sess.run([en_optim,self.loss],feed_dict={self.inputs:
batch_inputs,self.y:batch_labels})
        print('InceptionV3_Epoch: [%2d] [%4d/%4d] time: %4.4f loss: %.8f '\
              %(ae_epoch,i,batch_idx,time.time()-start_time,loss))
        #保存模型
        if counter%10==0:
            self.save(config.checkpoint_dir,counter)
            counter+=1
    def alexmodel(self,inputs):
        #=====
        #开始构建 alexnet 模型
        #=====
        #第一层卷积
        conv1=conv2d(inputs,64,k_h=11, k_w=11, d_h=4, d_w=4,name='conv1')
        #LRN 层和最大池化
        lrn1=tf.nn.lrn(conv1,4,bias=1.0,alpha=0.001/9,beta=0.75,name='lrn1')

        pool1=tf.nn.max_pool(lrn1,ksize=[1,3,3,1],strides=[1,2,2,1],padding='VALID',name='pool1')
        #第二层卷积
        conv2=conv2d(pool1,192,k_h=5, k_w=5, d_h=1, d_w=1,name='conv2')
        lrn2=tf.nn.lrn(conv2,4,bias=1.0,alpha=0.001/9,beta=0.75,name='lrn2')

        pool2=tf.nn.max_pool(lrn2,ksize=[1,3,3,1],strides=[1,2,2,1],padding='VALID',name='pool2')
        #第三层卷积
        conv3=conv2d(pool2,384,k_h=3, k_w=3, d_h=1, d_w=1,name='conv3')
        #第四层卷积
        conv4=conv2d(conv3,256,k_h=3, k_w=3, d_h=1, d_w=1,name='conv4')

```

```

#第五层卷积
conv5=conv2d(conv4,256,k_h=3, k_w=3, d_h=1, d_w=1,name='conv5')
#最大池化层

pool5=tf.nn.max_pool(conv5,ksize=[1,3,3,1],strides=[1,2,2,1],padding='VALID',name='pool5')
#三个全连接层
fc6=linear(tf.reshape(pool5,[self.batch_size,-1]),4096,name='fc6')
fc7=linear(fc6,4096,name='fc7')
fc8=linear(fc7,1000,relu=False,name='fc8')

return fc8

@property
def model_dir(self):
    return '{}_{}'.format(self.dataset_name,self.batch_size)
def save(self,checkpoint_dir,step):
    model_name='InceptionV3.model'
    checkpoint_dir=os.path.join(checkpoint_dir,self.model_dir)
    if not os.path.exists(checkpoint_dir):

        os.makedirs(checkpoint_dir)
    self.saver.save(self.sess,os.path.join(checkpoint_dir,model_name),
global_step=step)

#定义一些超参数
flags=tf.app.flags
flags.DEFINE_integer("epoch", 5, "Epoch to train [25]")
flags.DEFINE_float("learning_rate", 0.0001, "Learning rate of for adam [0.0002]")
flags.DEFINE_float("beta1", 0.1, "Momentum term of adam [0.5]")
flags.DEFINE_integer("train_size", 10, "The size of train images [np.inf]")
flags.DEFINE_integer("batch_size", 32, "The size of batch images [64]")
flags.DEFINE_string("dataset", "XXX.tfrecords", "The name of dataset")
flags.DEFINE_string("checkpoint_dir", "checkpoints", "Directory name to save the
checkpoints [checkpoint]")
FLAGS=flags.FLAGS

if __name__ == '__main__':

    with tf.Session() as sess:
        alexnet = AlexNet(
            sess,

```



```
batch_size=FLAGS.batch_size,  
dataset_name=FLAGS.dataset,  
checkpoint_dir=FLAGS.checkpoint_dir)  
alexnet.train(FLAGS)
```

5.5.3 Inception 模型

自 2012 年 AlexNet 拿到 ImageNet 的冠军并对神经网络有着历史性的促进意义后，神经网络模型的主要思路便开始向着更深层的神经网络进行突破，但这又给如何避免过拟合及计算量的增大带来了极大的考验。

Inception-V1 模型

为了既减小计算量又避免过拟合，谷歌的研发团队经过长时间研究后提出了 Inception 模型的第一版，也叫作 GoogleNet 模型。

图 5-17 为 GoogleNet 模型的结构图。

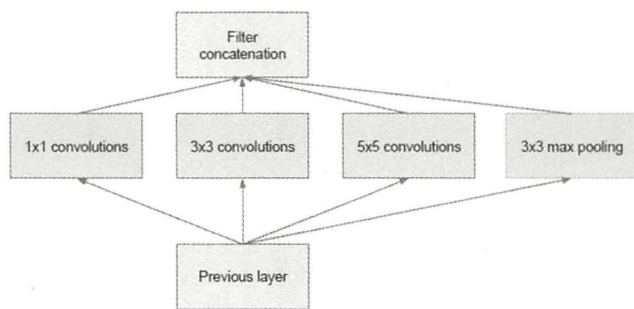


图 5-17

在这个模型中，首先将卷积核进行了分组的处理，所谓的分组，其实就是将其处理成了 1×1 , 3×3 , 5×5 三种规格的卷积核，并对 3×3 的卷积核进行了池化操作，其目的是降低其维度，使计算起来更加容易，这个版本也是最原始的 Inception 模型。在这里需要注意的是，所用的 1×1 , 3×3 , 5×5 三种卷积核只是示意，在实际操作的时候并不是必需的。

这个模型作为 Inception 模型的第一个基础版本，其优点就是实现了稀疏减少参

数的效果，但是缺点也非常明显，即每一层的卷积都是在上一层输出的基础上所得来的，这就使得在进行最后一层卷积，也就是 5×5 卷积的时候所需要的计算量就变得非常大，因此，谷歌对其模型进行了相应的改善，从而有了如图 5-18 所示的模型，这也是谷歌对外发布的第一个正式版本的 Inception-V1 模型。

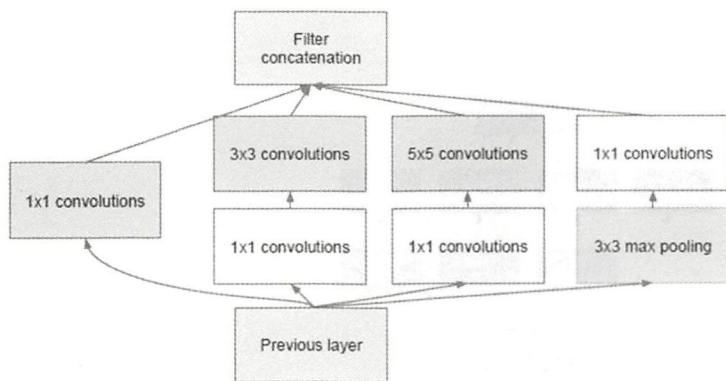


图 5-18

这一版本的 Inception 模型相比最初的模型而言，最大的一个变化就是在进行 3×3 和 5×5 的卷积之前进行了一次池化操作，其目的就是为了降低其维度，然后再将得到的结果进行下一次的卷积，从而使得计算量下降，所得到的结果变得更加优化。

下面我们来讲解一下 Inception V1 的整体结果，如图 5-19 所示。

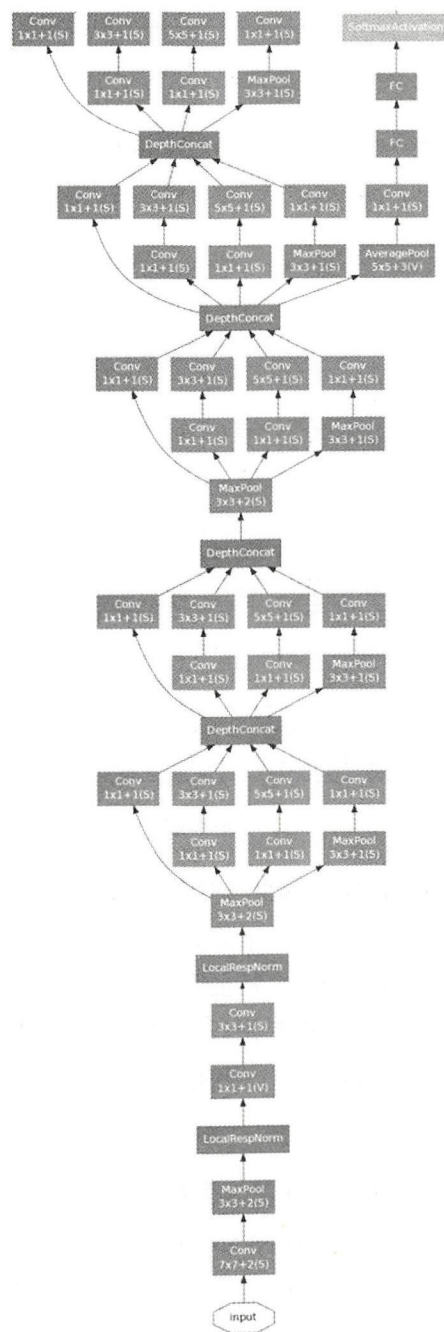


图 5-19

由于该模型比较长，所以更清楚的模型可以参考官网原图。下面我们对这个模型做相对详细的解释。

为了更好地学习和理解这个模型，可以根据官网的参数进行对照，其参数详情如图 5-20 所示。

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

图 5-20

(1) Inception-V1 (GoogleNet) 模型一共有 22 层，原始输入数据大小为 $224 \times 224 \times 3$ ，在进行第一层卷积 conv1 的时候，其 pad 为 3 (pad 表示对图像进行扩边，防止在卷积操作中丢失边界特征)，包括 64 个特征图，卷积核大小为 7×7 ，步长为 2，输出的特征图大小为 $112 \times 112 \times 64$ ，然后对其进行一次 ReLU 操作，使用 3×3 、步长为 2 的内核进行池化操作 (pool1) 后，得到一个边为 $[(112-3+1)/2]+1=56$ 特征图，大小为 56×56 、个数为 64 个。

(2) 进行第二层卷积 conv2，在第二层卷积中，pad 为 1，卷积核大小为 3×3 ，包括 192 个特征图，输出的特征图大小为 $56 \times 56 \times 192$ ，然后进行 ReLU 操作，使用 3×3 ，步长为 2 的内核进行池化操作 (pool2) 后，得到一个边为 $[(56-3+1)/2]+1=28$ 的特征图，大小为 28×28 、个数为 192 个，然后将这些特征图分成 4 条支线。



- 第一条支线：使用 64 个 1×1 的卷积核进行卷积，然后对卷积后的结果进行 ReLU 计算，得到一个大小为 28×28 、个数为 64 个的特征图。
- 第二条支线：使用 96 个 1×1 的卷积核作为 3×3 卷积核之前的“reduce”，变成一个大小为 28×28 、个数为 96 的特征图，对其进行 ReLU 计算后，再使用 128 个大小为 3×3 的内核进行池化操作，得到大小为 28×28 、个数为 128 的特征图。
- 第三条支线：使用 16 个 1×1 的卷积核作为 5×5 卷积核之前的“reduce”，变成一个大小为 28×28 、个数为 16 的特征图，对其进行 ReLU 计算后再使用 32 个大小为 5×5 、pad 为 2 的内核进行池化操作，得到 28×28 大小、个数为 32 的特征图。
- 第四条支线：该支线是一个池化层，其池化内核的大小为 3×3 、pad 为 1，输出还是大小为 28×28 、个数为 192 的特征图，然后使用 32 个 1×1 的内核进行池化操作，最后输出大小为 28×28 、个数为 256 的特征图。

(3) 最后，将这 4 个结果进行连接，其输出为 $28 \times 28 \times 256$ 。

(4) 将其结果再分成 4 条支线，进行 3b 的“inception module”，其操作过程与 3a 类似，可以通过图 5-20 进行推导，最后得到大小为 28×28 、个数为 64 的特征图。

(5) 再将这 4 个结果进行连接，其输出为 $28 \times 28 \times 480$ 。

重复上述过程，根据图 5-20，可以计算出 4a 到 5b 的计算结果。

这里值得注意的是，为了避免梯度的消失，在这个模型中额外增加了两个辅助的 Softmax 函数用于向前传导梯度。

Inception-V1 模型采用了模块化的结果，并使用 average pooling 代替了全连接层，还保留了 Dropout 层，并且使用了两个辅助的 Softmax 函数来避免梯度消失，这不仅提升了预测分类的准确率，而且极大地减少了参数的数量，使其在实际训练过程中获得了巨大的成功。

Inception-V2 模型

在 Inception-V1 模型取得了巨大的成功之后，谷歌对其进行了改进，推出了第二个版本——Inception-V2 模型。相比于第一个版本而言，第二个版本的 Inception 模型



更加接近神经网络在实际生活中的需求。在这个版本中，其最大的改进就是增加了 BN（Batch Normalization，批规范化）层。BN 层的主要作用有以下四点：

- （1）加速收敛；
- （2）防止过拟合，可以少用甚至不用正则表达式和 Dropout 函数；
- （3）降低网络对初始化权重的不敏感；
- （4）允许使用较大的学习率。

另一方面，V2 模型使用了 2 个 3×3 的卷积代替了 V1 模型中的 5×5 ，这样做不仅降低了参数的数量，同时也使计算速度变得更快，如图 5-21 所示。

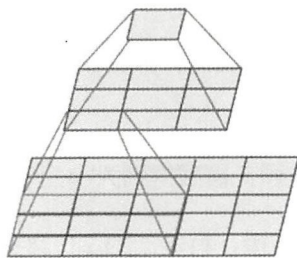


图 5-21

Inception-V2 模型的网络参数如图 5-22 所示。

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	double #3×3 reduce	double #3×3	Pool +proj
convolution*	7×7/2	112×112×64	1						
max pool	3×3/2	56×56×64	0						
convolution	3×3/1	56×56×192	1		64	192			
max pool	3×3/2	28×28×192	0						
inception (3a)		28×28×256	3	64	64	64	64	96	avg + 32
inception (3b)		28×28×320	3	64	64	96	64	96	avg + 64
inception (3c)	stride 2	28×28×576	3	0	128	160	64	96	max + pass through
inception (4a)		14×14×576	3	224	64	96	96	128	avg + 128
inception (4b)		14×14×576	3	192	96	128	96	128	avg + 128
inception (4c)		14×14×576	3	160	128	160	128	160	avg + 128
inception (4d)		14×14×576	3	96	128	192	160	192	avg + 128
inception (4e)	stride 2	14×14×1024	3	0	128	192	192	256	max + pass through
inception (5a)		7×7×1024	3	352	192	320	160	224	avg + 128
inception (5b)		7×7×1024	3	352	192	320	192	224	max + 128
avg pool	7×7/1	1×1×1024	0						

图 5-22



Inception-V3 模型

经历了 Inception-V2 模型后，谷歌又推出了其改进版本 Inception-V3 模型。V3 模型相对于前两个模型而言最大的改进就是将之前的 7×7 分解成了两个一维卷积 (1×7 和 7×1)，将之前的 3×3 同样也分解成了两个一维卷积 (1×3 和 3×1)。众所周知，当维数越低的时候，其计算速度就会变得越快，这就腾出计算能力用于加深网络的相关操作；另一方面，值得注意的是，在 V3 模型中，网络输入也发生了变化，从之前的 224×224 变成了 299×299 。Inception-V3 模型的参数如图 5-23 所示。

type	patch size/stride or remarks	input size
conv	$3 \times 3/2$	$299 \times 299 \times 3$
conv	$3 \times 3/1$	$149 \times 149 \times 32$
conv padded	$3 \times 3/1$	$147 \times 147 \times 32$
pool	$3 \times 3/2$	$147 \times 147 \times 64$
conv	$3 \times 3/1$	$73 \times 73 \times 64$
conv	$3 \times 3/2$	$71 \times 71 \times 80$
conv	$3 \times 3/1$	$35 \times 35 \times 192$
3×Inception	As in figure 5	$35 \times 35 \times 288$
5×Inception	As in figure 6	$17 \times 17 \times 768$
2×Inception	As in figure 7	$8 \times 8 \times 1280$
pool	8×8	$8 \times 8 \times 2048$
linear	logits	$1 \times 1 \times 2048$
softmax	classifier	$1 \times 1 \times 1000$

图 5-23

下面使用 TensorFlow 来实现一个 Inception-V3 模型：

```
import tensorflow as tf
slim=tf.contrib.slim
import os
import time

trunc_normal=lambda stddev: tf.truncated_normal_initializer(0.0,stddev)
height,width=299,299

class InceptionV3(object):
    def
__init__(self,sess,batch_size=64,dataset_name='default',checkpoint_dir=None):
    self.y_dim=1000
    self.batch_size=batch_size
```


TensorFlow 进阶指南

基础、算法与应用

```

        self.dataset_name=dataset_name
        self.checkpoint_dir=checkpoint_dir
        self.frame_size=3
        self.build_model()
        self.sess=sess

def build_model(self):
    #构建模型
    image_dims=[height,width,self.frame_size]
    y_dim=[self.y_dim]

    #定义网络入口
    self.inputs=tf.placeholder(tf.float32,[self.batch_size]+image_dims,name='real_images')
    self.y=tf.placeholder(tf.float32,[self.batch_size]+y_dim,name='y')

    #定义 forward 和 loss
    self.y_logits,self.end_points=self.inception_v3(self.inputs,is_training=True)
    self.loss=tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=self.y,logits=self.y_logits))

    #保存模型
    self.saver=tf.train.Saver()
def train(self,config):
    #定义优化器
    en_optim=tf.train.AdamOptimizer(config.learning_rate,beta1=config.beta1).
    minimize(self.loss)

    try:
        tf.global_variables_initializer().run()
    except:
        tf.initialize_all_variables().run()

    start_time=time.time()

    #定义计数器
    counter=0

    #开始迭代训练
    print('start training...')
    for ae_epoch in range(config.epoch):

```



```

batch_idx=config.train_size

for i in range(batch_idx):
    #定义输入, 这里以随机数代替, 读者根据自身需求更改输入
    inputs=tf.random_uniform((self.batch_size,height,width,3))
    labels=tf.one_hot(tf.constant(1,shape=[self.batch_size]),depth
=1000,on_value=1.0,off_value=0.0)
    #这里需要先从定义的 inputs 和 labels 等张量得到具体的数字
    batch_inputs,batch_labels=sess.run([inputs,labels])

    _,loss=self.sess.run([en_optim,self.loss],feed_dict={self.inpu
ts:batch_inputs,self.y:batch_labels})
    print('InceptionV3_Epoch: [%2d][%4d/%4d] time: %4.4f loss: %.8f '\
        %(ae_epoch,i,batch_idx,time.time()-start_time,loss))
    #保存模型
    if counter%10==0:
        self.save(config.checkpoint_dir,counter)
        counter+=1

def inception_v3_arg_scope(self,weight_decay=0.00004,stddev=0.1,batch_
norm_var_collection='moving_vars'):
    #=====
    #该函数用来生成网络中经常用到的函数的默认参数, 如卷积的激活函数、权重初始化方式、标准
    化器等
    #=====
    batch_norm_params={
        'decay':0.9997,
        'epsilon':0.001,
        'updates_collections':tf.GraphKeys.UPDATE_OPS,
        'variables_collections':{
            'beta':None,
            'gamma':None,
            'moving_mean':[batch_norm_var_collection],
            'moving_variance':[batch_norm_var_collection]
        }
    }

    #slim.arg_scope 是一个非常有用的工具, 可以给函数的参数自动赋予某些默认值, 使得后面定
    义卷积层变得非常方便

    with slim.arg_scope([slim.conv2d,slim.fully_connected],weights_
regularizer=slim.l2_regularizer(weight_decay)):
        with slim.arg_scope(
            [slim.conv2d],

```




```

        weights_initializer=tf.truncated_normal_initializer(stddev=stddev),
        activation_fn=tf.nn.relu,
        normalizer_fn=slim.batch_norm,
        normalizer_params=batch_norm_params) as sc:
            return sc
def inception_v3_base(self,inputs,scope=None):
    #=====
    #该函数用于生成 Inception-v3 网络的卷积部分
    #=====

    #定义一个字典 end_points, 用于保存某些关键节点, 供之后使用
    end_points={}
    with tf.variable_scope(scope,'InceptionV3',[inputs]) as scope:
        with slim.arg_scope([slim.conv2d,slim.max_pool2d,slim.avg_pool2d],
stride=1,padding='VALID'):
            net=slim.conv2d(inputs,32,[3,3],stride=2,scope='Conv2d_1a_3x3')
            net=slim.conv2d(net,32,[3,3],scope='Conv2d_2a_3x3')
            net=slim.conv2d(net,64,[3,3],padding='SAME',scope='Conv2d_2b_3x3')
            net=slim.max_pool2d(net,[3,3],stride=2,scope='MaxPool_3a_3x3')
            net=slim.conv2d(net,80,[1,1],scope='Conv2d_3b_1x1')
            net=slim.conv2d(net,192,[3,3],scope='Conv2d_4a_3x3')
            net=slim.max_pool2d(net,[3,3],stride=2,scope='MaxPool_5a_3x3')
            #接下来是连续三个 Inception 模块组, 这三个模块组分别有多个 Inception Module.
            #Inception block
            with slim.arg_scope([slim.conv2d,slim.max_pool2d,slim.avg_pool2d],
stride=1,padding='SAME'):
                #first group inceptions
                with tf.variable_scope('Mixed_5b'):
                    with tf.variable_scope('Branch_0'):
                        branch_0=slim.conv2d(net,64,[1,1],scope='Conv2d_0a_1x1')
                    with tf.variable_scope('Branch_1'):
                        branch_1=slim.conv2d(net,48,[1,1],scope='Conv2d_0a_1x1')
                        branch_1=slim.conv2d(branch_1,64,[5,5],scope='Conv2d_0b_5x5')
                    with tf.variable_scope('Branch_2'):
                        branch_2 = slim.conv2d(net, 64, [1, 1], scope='Conv2d_0a_1x1')
                        branch_2 = slim.conv2d(branch_2, 96, [3,
3],scope='Conv2d_0b_3x3')
                        branch_2 = slim.conv2d(branch_2, 96, [3,
3],scope='Conv2d_0c_3x3')
                    with tf.variable_scope('Branch_3'):
                        branch_3 = slim.avg_pool2d(net, [3, 3],
scope='AvgPool_0a_3x3')

```



```
        branch_3 = slim.conv2d(branch_3, 32, [1,
1],scope='Conv2d_0b_1x1')
        net = tf.concat([branch_0, branch_1, branch_2, branch_3],3)
    with tf.variable_scope('Mixed_5c'):
        with tf.variable_scope('Branch_0'):
            branch_0=slim.conv2d(net,64,[1,1],scope='Conv2d_0a_1x1')
        with tf.variable_scope('Branch_1'):
            branch_1=slim.conv2d(net,48,[1,1],scope='Conv2d_0b_1x1')
            branch_1=slim.conv2d(branch_1,64,[5,5],scope='Conv_1_0c_5x5')
        with tf.variable_scope('Branch_2'):
            branch_2 = slim.conv2d(net, 64, [1, 1], scope='Conv2d_0a_1x1')
            branch_2 = slim.conv2d(branch_2, 96, [3, 3],
scope='Conv2d_0b_3x3')
            branch_2 = slim.conv2d(branch_2, 96, [3, 3],
scope='Conv2d_0c_3x3')
        with tf.variable_scope('Branch_3'):
            branch_3 = slim.avg_pool2d(net, [3, 3],
scope='AvgPool_0a_3x3')
            branch_3 = slim.conv2d(branch_3, 64, [1, 1],
scope='Conv2d_0b_1x1')
        net = tf.concat([branch_0, branch_1, branch_2, branch_3],3)
    with tf.variable_scope('Mixed_5d'):
        with tf.variable_scope('Branch_0'):
            branch_0=slim.conv2d(net,64,[1,1],scope='Conv2d_0a_1x1')
        with tf.variable_scope('Branch_1'):
            branch_1=slim.conv2d(net,48,[1,1],scope='Conv2d_0a_1x1')
            branch_1=slim.conv2d(branch_1,64,[5,5],scope='Conv2d_0b_5x5')
        with tf.variable_scope('Branch_2'):
            branch_2 = slim.conv2d(net, 64, [1, 1], scope='Conv2d_0a_1x1')
            branch_2 = slim.conv2d(branch_2, 96, [3, 3],
scope='Conv2d_0b_3x3')
            branch_2 = slim.conv2d(branch_2, 96, [3, 3],
scope='Conv2d_0c_3x3')
        with tf.variable_scope('Branch_3'):
            branch_3 = slim.avg_pool2d(net, [3, 3],
scope='AvgPool_0a_3x3')
            branch_3 = slim.conv2d(branch_3, 64, [1, 1],
scope='Conv2d_0b_1x1')
        net = tf.concat([branch_0, branch_1, branch_2, branch_3],3)
    #second group inceptions
    with tf.variable_scope('Mixed_6a'):
        with tf.variable_scope('Branch_0'):
            branch_0=slim.conv2d(net,384,[3,3],stride=2,padding='VALID'
```



```

,scope='Conv2d_1a_1x1')
    with tf.variable_scope('Branch_1'):
        branch_1=slim.conv2d(net,64,[1,1],scope='Conv2d_0a_1x1')
        branch_1=slim.conv2d(branch_1,96,[3,3],scope='Conv2d_0b_3x3')
        branch_1=slim.conv2d(branch_1,96,[3,3],stride=2,padding='VA
LID',scope='Conv2d_1a_1x1')
    with tf.variable_scope('Branch_2'):
        branch_2 = slim.max_pool2d(net, [3, 3],
stride=2,padding='VALID',scope='MaxPool_1a_3x3')
        net = tf.concat([branch_0, branch_1, branch_2],3)

with tf.variable_scope('Mixed_6b'):
    with tf.variable_scope('Branch_0'):
        branch_0=slim.conv2d(net,192,[1,1],scope='Conv2d_0a_1x1')
    with tf.variable_scope('Branch_1'):
        branch_1=slim.conv2d(net,128,[1,1],scope='Conv2d_0a_1x1')
        branch_1=slim.conv2d(branch_1,128,[1,7],scope='Conv2d_0b_1x7')
        branch_1=slim.conv2d(branch_1,192,[7,1],scope='Conv2d_0c_7x1')
    with tf.variable_scope('Branch_2'):
        branch_2 = slim.conv2d(net, 128, [1, 1], scope='Conv2d_0a_1x1')
        branch_2 = slim.conv2d(branch_2, 128, [7,1],scope='Conv2d_0b_7x1')
        branch_2 = slim.conv2d(branch_2, 128, [1,7],scope='Conv2d_0c_1x7')
        branch_2 = slim.conv2d(branch_2, 128, [7,1],scope='Conv2d_0d_7x1')
        branch_2 = slim.conv2d(branch_2, 192, [1,7],scope='Conv2d_0e_1x7')
    with tf.variable_scope('Branch_3'):
        branch_3 = slim.avg_pool2d(net, [3, 3], scope='AvgPool_0a_3x3')
        branch_3 = slim.conv2d(branch_3, 192, [1, 1],scope='Conv2d_0b_1x1')
        net = tf.concat([branch_0, branch_1, branch_2, branch_3],3)

with tf.variable_scope('Mixed_6c'):
    with tf.variable_scope('Branch_0'):
        branch_0=slim.conv2d(net,192,[1,1],scope='Conv2d_0a_1x1')
    with tf.variable_scope('Branch_1'):
        branch_1=slim.conv2d(net,160,[1,1],scope='Conv2d_0a_1x1')
        branch_1=slim.conv2d(branch_1,160,[1,7],scope='Conv2d_0b_1x7')
        branch_1=slim.conv2d(branch_1,192,[7,1],scope='Conv2d_0c_7x1')
    with tf.variable_scope('Branch_2'):
        branch_2 = slim.conv2d(net, 160, [1, 1], scope='Conv2d_0a_1x1')
        branch_2 = slim.conv2d(branch_2, 160, [7,1],scope='Conv2d_0b_7x1')
        branch_2 = slim.conv2d(branch_2, 160, [1,7],scope='Conv2d_0c_1x7')
        branch_2 = slim.conv2d(branch_2, 160, [7,1],scope='Conv2d_0d_7x1')
        branch_2 = slim.conv2d(branch_2, 192, [1,7],scope='Conv2d_0e_1x7')
    with tf.variable_scope('Branch_3'):

```

```
        branch_3 = slim.avg_pool2d(net, [3, 3], scope='AvgPool_0a_3x3')
        branch_3 = slim.conv2d(branch_3, 192, [1, 1], scope='Conv2d_0b_1x1')
    net = tf.concat([branch_0, branch_1, branch_2, branch_3], 3)

    with tf.variable_scope('Mixed_6d'):
        with tf.variable_scope('Branch_0'):
            branch_0 = slim.conv2d(net, 192, [1, 1], scope='Conv2d_0a_1x1')
        with tf.variable_scope('Branch_1'):
            branch_1 = slim.conv2d(net, 160, [1, 1], scope='Conv2d_0a_1x1')
            branch_1 = slim.conv2d(branch_1, 160, [1, 7], scope='Conv2d_0b_1x7')
            branch_1 = slim.conv2d(branch_1, 192, [7, 1], scope='Conv2d_0c_7x1')
        with tf.variable_scope('Branch_2'):
            branch_2 = slim.conv2d(net, 160, [1, 1], scope='Conv2d_0a_1x1')
            branch_2 = slim.conv2d(branch_2, 160, [7, 1], scope='Conv2d_0b_7x1')
            branch_2 = slim.conv2d(branch_2, 160, [1, 7], scope='Conv2d_0c_1x7')
            branch_2 = slim.conv2d(branch_2, 160, [7, 1], scope='Conv2d_0d_7x1')
            branch_2 = slim.conv2d(branch_2, 192, [1, 7], scope='Conv2d_0e_1x7')
        with tf.variable_scope('Branch_3'):
            branch_3 = slim.avg_pool2d(net, [3, 3], scope='AvgPool_0a_3x3')
            branch_3 = slim.conv2d(branch_3, 192, [1, 1], scope='Conv2d_0b_1x1')
    net = tf.concat([branch_0, branch_1, branch_2, branch_3], 3)

    with tf.variable_scope('Mixed_6e'):
        with tf.variable_scope('Branch_0'):
            branch_0 = slim.conv2d(net, 192, [1, 1], scope='Conv2d_0a_1x1')
        with tf.variable_scope('Branch_1'):
            branch_1 = slim.conv2d(net, 192, [1, 1], scope='Conv2d_0a_1x1')
            branch_1 = slim.conv2d(branch_1, 192, [1, 7], scope='Conv2d_0b_1x7')
            branch_1 = slim.conv2d(branch_1, 192, [7, 1], scope='Conv2d_0c_7x1')
        with tf.variable_scope('Branch_2'):
            branch_2 = slim.conv2d(net, 192, [1, 1], scope='Conv2d_0a_1x1')
            branch_2 = slim.conv2d(branch_2, 192, [7, 1], scope='Conv2d_0b_7x1')
            branch_2 = slim.conv2d(branch_2, 192, [1, 7], scope='Conv2d_0c_1x7')
            branch_2 = slim.conv2d(branch_2, 192, [7, 1], scope='Conv2d_0d_7x1')
            branch_2 = slim.conv2d(branch_2, 192, [1, 7], scope='Conv2d_0e_1x7')
        with tf.variable_scope('Branch_3'):
            branch_3 = slim.avg_pool2d(net, [3, 3], scope='AvgPool_0a_3x3')
            branch_3 = slim.conv2d(branch_3, 192, [1, 1], scope='Conv2d_0b_1x1')
    net = tf.concat([branch_0, branch_1, branch_2, branch_3], 3)
    end_points['Mixed_6e'] = net

    #third group inceptions
    with tf.variable_scope('Mixed_7a'):
        with tf.variable_scope('Branch_0'):
```



```

        branch_0=slim.conv2d(net,192,[1,1],scope='Conv2d_0a_1x1')
        branch_0=slim.conv2d(branch_0,320,[3,3],stride=2,padding='V
ALID',scope='Conv2d_1a_3x3')
        with tf.variable_scope('Branch_1'):
            branch_1=slim.conv2d(net,192,[1,1],scope='Conv2d_0a_1x1')
            branch_1=slim.conv2d(branch_1,192,[1,7],scope='Conv2d_0b_1x7')
            branch_1=slim.conv2d(branch_1,192,[7,1],scope='Conv2d_0c_7x1')
            branch_1=slim.conv2d(branch_1,192,[3,3],stride=2,padding='V
ALID',scope='Conv2d_1a_3x3')
        with tf.variable_scope('Branch_2'):
            branch_2 = slim.max_pool2d(net, [3, 3],
stride=2,padding='VALID',scope='MaxPool_1a_3x3')
            net = tf.concat([branch_0, branch_1, branch_2],3)
        with tf.variable_scope('Mixed_7b'):
            with tf.variable_scope('Branch_0'):
                branch_0=slim.conv2d(net,320,[1,1],scope='Conv2d_0a_1x1')
            with tf.variable_scope('Branch_1'):
                branch_1=slim.conv2d(net,384,[1,1],scope='Conv2d_0a_1x1')
                branch_1=tf.concat([
                    slim.conv2d(branch_1,384,[1,3],scope='Conv2d_0b_1x3'),
                    slim.conv2d(branch_1,384,[3,1],scope='Conv2d_0b_3x1')],3)
            with tf.variable_scope('Branch_2'):
                branch_2 = slim.conv2d(net, 448, [1, 1], scope='Conv2d_0a_1x1')
                branch_2 = slim.conv2d(branch_2, 384,
[3,3],scope='Conv2d_0b_3x3')
                branch_2=tf.concat([
                    slim.conv2d(branch_2,384,[1,3],scope='Conv2d_0c_1x3'),
                    slim.conv2d(branch_2,384,[3,1],scope='Conv2d_0d_3x1')],3)
            with tf.variable_scope('Branch_3'):
                branch_3 = slim.avg_pool2d(net, [3, 3],
scope='AvgPool_0a_3x3')
                branch_3 = slim.conv2d(branch_3, 192, [1, 1],scope='Conv2d_0b_1x1')
            net = tf.concat([branch_0, branch_1, branch_2, branch_3],3)
        with tf.variable_scope('Mixed_7c'):
            with tf.variable_scope('Branch_0'):
                branch_0=slim.conv2d(net,320,[1,1],scope='Conv2d_0a_1x1')
            with tf.variable_scope('Branch_1'):
                branch_1=slim.conv2d(net,384,[1,1],scope='Conv2d_0a_1x1')
                branch_1=tf.concat([
                    slim.conv2d(branch_1,384,[1,3],scope='Conv2d_0b_1x3'),
                    slim.conv2d(branch_1,384,[3,1],scope='Conv2d_0c_3x1')],3)
            with tf.variable_scope('Branch_2'):
                branch_2 = slim.conv2d(net, 448, [1, 1], scope='Conv2d_0a_1x1')

```

```
        branch_2 = slim.conv2d(branch_2, 384, [3, 3], scope='Conv2d_0b_3x3')
        branch_2 = tf.concat([
            slim.conv2d(branch_2, 384, [1, 3], scope='Conv2d_0c_1x3'),
            slim.conv2d(branch_2, 384, [3, 1], scope='Conv2d_0d_3x1')], 3)
        with tf.variable_scope('Branch_3'):
            branch_3 = slim.avg_pool2d(net, [3, 3], scope='AvgPool_0a_3x3')
            branch_3 = slim.conv2d(branch_3, 192, [1, 1], scope='Conv2d_0b_1x1')
        net = tf.concat([branch_0, branch_1, branch_2, branch_3], 3)
    return net, end_points

def inception_v3(self, inputs,
                num_classes=1000,
                is_training=True,
                dropout_keep_prob=0.8,
                prediction_fn=slim.softmax,
                spatial_squeeze=True,
                reuse=None,
                scope='InceptionV3'):
    #=====
    #该函数主要包括全局平均池化、softmax和Auxiliary Logist
    #=====
    with tf.variable_scope(scope, 'InceptionV3', [inputs, num_classes],
        reuse=reuse) as scope:
        with slim.arg_scope([slim.batch_norm, slim.dropout],
            is_training=is_training):
            net, end_points = self.inception_v3_base(inputs, scope=scope)
            #接下来是Auxiliary Logist 辅助节点分类
            with slim.arg_scope([slim.conv2d, slim.max_pool2d, slim.avg_pool2d],
                stride=1, padding='SAME'):
                aux_logist = end_points['Mixed_6e']
                with tf.variable_scope('AuxLogist'):
                    aux_logist = slim.avg_pool2d(aux_logist, [5, 5], stride=3, padding='
VALID', scope='AvgPool_1a_5x5')
                    aux_logist = slim.conv2d(aux_logist, 128, [1, 1], scope='Conv2d_1b_1x1')
                    aux_logist = slim.conv2d(aux_logist, 768, [5, 5], weights_initializer
r=trunc_normal(0.01), padding='VALID', scope='Conv2d_2a_5x5')
                    aux_logist = slim.conv2d(aux_logist, num_classes, [1, 1], activation
_fn=None, normalizer_fn=None, weights_initializer=trunc_normal(0.01), scope='Conv
2d_2b_1x1')

                if spatial_squeeze:
                    aux_logist = tf.squeeze(aux_logist, [1, 2], name='SpatialSqu
eeze')
```



```
        end_points['AuxLogist']=aux_logist
    #接下来是正常的分类预测逻辑
    with tf.variable_scope('Logist'):
        net=slim.avg_pool2d(net, [8,8],scope='AvgPool_1a_8x8')
        net=slim.dropout(net,keep_prob=dropout_keep_prob,scope='Dropout_1b')
        end_points['PreLogist']=net
        logist=slim.conv2d(net,num_classes,[1,1],activation_fn=None,normalizer_fn=None,scope='Conv2d_1c_1x1')
        if spatial_squeeze:
            logist=tf.squeeze(logist,[1,2],name='SpatialSqueeze')
        end_points['Logist']=logist
        end_points['Predictions']=prediction_fn(logist,scope='Predictions')
    return logist,end_points

@property
def model_dir(self):
    return '{}_{}'.format(self.dataset_name,self.batch_size)
def save(self,checkpoint_dir,step):
    model_name='InceptionV3.model'
    checkpoint_dir=os.path.join(checkpoint_dir,self.model_dir)
    if not os.path.exists(checkpoint_dir):
        os.makedirs(checkpoint_dir)
    self.saver.save(self.sess,os.path.join(checkpoint_dir,model_name),global_step=step)

#定义一些超参数
flags=tf.app.flags
flags.DEFINE_integer("epoch", 5, "Epoch to train [25]")
flags.DEFINE_float("learning_rate", 0.0001, "Learning rate of for adam [0.0002]")
flags.DEFINE_float("beta1", 0.1, "Momentum term of adam [0.5]")
flags.DEFINE_integer("train_size", 10, "The size of train images [np.inf]")
flags.DEFINE_integer("batch_size", 32, "The size of batch images [64]")
flags.DEFINE_string("dataset", "XXX.tfrecords", "The name of dataset")
flags.DEFINE_string("checkpoint_dir", "checkpoints", "Directory name to save the checkpoints [checkpoint]")
FLAGS=flags.FLAGS

if __name__ == '__main__':
    with tf.Session() as sess:
        inceptionv3 = InceptionV3(
```

```
sess,  
    batch_size=FLAGS.batch_size,  
    dataset_name=FLAGS.dataset,  
    checkpoint_dir=FLAGS.checkpoint_dir)  
inceptionv3.train(FLAGS)
```

5.6 用 CNN 实现 MNIST 训练

第 2 章中使用普通的线性模型训练出来的 MNIST 数据集准确率仅能达到 92% 左右, 然而对于一个成熟的数据集来讲, 92% 的准确率是远远不够的, 而线性模型也是一个极其简单的模型, 如果使用卷积神经网络进行训练, 其准确率将会达到 99% 以上, 下面用 CNN 来训练一个 MNIST 数据集, 步骤如下所述。

- (1) 导入 MNIST 数据集;
- (2) 定义输入和输出参数;
- (3) 定义卷积模型;
- (4) 定义训练模型;
- (5) 开始训练;
- (6) 验证训练结果。

接下来就按照以上 6 个步骤开始操作。首先要做的就是使用 `input_data.py` 文件导入数据集, 该文件可以在 `tensorflow/examples/tutorials/mnist` 下找到, 也可以自己来写, 这里使用作者自己编写的 `input_data.py` 文件进行操作。

`input_data.py` 文件:

```
from __future__ import absolute_import  
from __future__ import division  
from __future__ import print_function  
  
import gzip  
import os  
import tempfile
```

```
import numpy
from six.moves import urllib
from six.moves import xrange
import tensorflow as tf
from tensorflow.contrib.learn.python.learn.datasets.mnist import read_data_sets
```

该文件只有一个作用，就是将 MNIST 数据集从 TensorFlow 所提供的相关数据集中下载下来。

接下来，定义一个 `mnist.py` 文件，导入数据集和相关系统库。

```
import tensorflow as tf
import os
import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

然后定义输入参数、权重和偏置量等信息。

```
sess = tf.InteractiveSession()
x = tf.placeholder(tf.float32, shape=[None, 784], name='x-input')
y_ = tf.placeholder(tf.float32, shape=[None, 10], name='y-input')
W = tf.Variable(tf.zeros([784, 10]), name='weight')
b = tf.Variable(tf.zeros([10]), name='bias')
```

下面定义一些基本的函数，包括权重的参数定义函数、偏置量的参数定义函数、卷积核的定义函数、池化层的定义函数：

```
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)

def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
```

在这里，将权重参数初始化为进行卷积定义时所传进来的 `shape`，并使用

`tf.truncated_normal()`函数将其进行正态分布的初始化操作；然后使用 `tf.constant()`函数将偏置量定义为一个常量，接下来使用 `tf.nn.conv2d()`函数来定义一个二维卷积，该函数的主要作用就是在给定四维输入（`input`）和权重 W 的情况下计算二维卷积，在这里我们所传入的 `x` 为四维输入；最后使用 `tf.nn.max_pool()`函数定义一个池化层，这里所使用的池化函数就是前面章节中所讲到的最大值池化函数。

当定义好基本的参数后，接下来，也是最重要的一步就是定义卷积模型。通过前面的讲解我们知道，卷积模型一般由卷积层、池化层、全连接层构成，为了防止过拟合现象的发生，我们还会在最后加一个 `Dropout` 层来随机丢弃一些数据。在本例中，我们定义 2 层卷积层、2 层池化层和 2 层全连接层，并通过 `Softmax` 函数作为激活函数将其激活：

```
W_conv1 = weight_variable([5,5,1,32])
b_conv1 = bias_variable([32])

x_image = tf.reshape(x, [-1,28,28,1])

h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)

W_conv2 = weight_variable([5,5,32,64])
b_conv2 = bias_variable([64])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)

W_fc1 = weight_variable([7 * 7 * 64,1024])
b_fc1 = bias_variable([1024])

h_pool2_flat = tf.reshape(h_pool2, [-1,7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

y_conv = tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
```


从上面的代码可以发现，我们在这里定义了权重值的大小、输入图像的大小，以及偏置量，并将它们传入定义好的函数中，然后使用 ReLU 函数进行相关的转化，并使用 Dropout 函数随机丢弃一些数据，生成所需要的参数。

最后，就可以定义交叉熵、训练步长、优化器和训练模型了：

```
with tf.name_scope('cross'):
    cross_entropy = -tf.reduce_sum(y_*tf.log(y_conv))
with tf.name_scope('train'):
    train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
with tf.name_scope('test'):
    correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(20000):
        batch = mnist.train.next_batch(50)
        if i % 100 == 0:

            train_accuracy = accuracy.eval(feed_dict={x: batch[0], y_:batch[1],
keep_prob: 1.0})
            print("step %d,training accuracy %g"%(i, train_accuracy))
            train_step.run(feed_dict = {x: batch[0], y_:batch[1], keep_prob: 0.5})

        print("test accuracy %g" %accuracy.eval(feed_dict={x: mnist.test.images, y_:
mnist.test.labels, keep_prob: 1.0}))
```

经过 20 000 步的训练，最后得到的准确率为 99.2%，如图 5-24 所示。

```
step 18600, training accuracy 1
step 18700, training accuracy 1
step 18800, training accuracy 1
step 18900, training accuracy 1
step 19000, training accuracy 1
step 19100, training accuracy 1
step 19200, training accuracy 1
step 19300, training accuracy 1
step 19400, training accuracy 1
step 19500, training accuracy 1
step 19600, training accuracy 1
step 19700, training accuracy 0.99
step 19800, training accuracy 1
step 19900, training accuracy 1
2016-06-03 17:31:23.510730: W C:\tf_jenkins\home\workspace\tf\win\X\windows-gpu\PY136\tensorflow\core\common_runtime\tf_allocator.cc:217] Allocator: GPU_0_hic
test accuracy 0.992
Process finished with exit code 0
```

图 5-24

第 6 章

循环神经网络

第 5 章着重讲解了卷积神经网络的相关知识和常用网络模型,并用 TensorFlow 程序将其一一实现,学以致用。

根据之前的章节,我们知道了卷积神经网络在图像处理、识别及分类方面有着非常明显的优势,甚至可以说卷积神经网络就是为图像的识别和处理而生的。但是在现实生活中,我们所要处理的内容不仅仅是图像,诸如智能客服、聊天机器人、在线实时翻译等自然语言处理 (Natural Language Processing, NLP) 类的相关需求也很多。为了使深度学习能够在自然语言处理方面表现得更加突出,人们提出了循环神经网络 (Recurrent Neural Networks, RNN)。

6.1 初识循环神经网络

循环神经网络是 20 世纪 80 年代末由 Jordan, Pineda, Williams, Elman 等神经网络专家提出的一种神经网络结构模型,最大特点是在处理单元之间既有内部反馈又有前馈连接。它实际上是一个反馈动力系统,具有更强的动态行为和计算能力。循环神经网络是目前深度学习领域中最流行的网络模型之一,在自然语言处理领域中发挥着不可替代的作用。

那么，到底什么是循环神经网络？循环神经网络又能做什么呢？

在详细地介绍循环神经网络之前，先来简单地介绍一下前馈神经网络、时序信息及其处理思想。

6.1.1 前馈神经网络

在前馈神经网络中，各个神经元分层排列且每个神经元只与前一层的神经元相连，接收前一层的输出，并传递给下一层，直到输出层，各层之间没有任何反馈，可以用一个有向且无环的图来表示。可以将每个神经元理解成一个感知器，只不过使用的激活函数是不同的。

我们来回顾一下感知器结构，如图 6-1 所示。

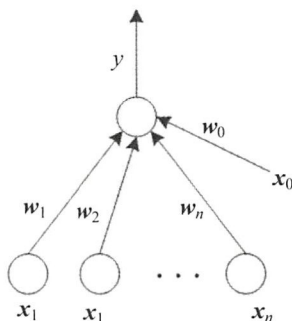


图 6-1

图 6-1 是一个传统的感知器，感知器中的激活函数将求和的结果和某个阈值比较大小，然后再决定其输出的值是什么。在传统的神经网络中，一般使用 Sigmoid 函数作为激活函数。通过前面章节的讲解可知，Sigmoid 函数具有中间增益而两侧抑制的特性，对于不同的区域信号有着不同的反应。

那么前馈神经网络又是怎样的呢？在这里以最典型的前馈神经网络——BP(Back Propagation, 也叫反向传播算法)神经网络为例，做一个简单的介绍。BP 神经网络的结构如图 6-2 所示。

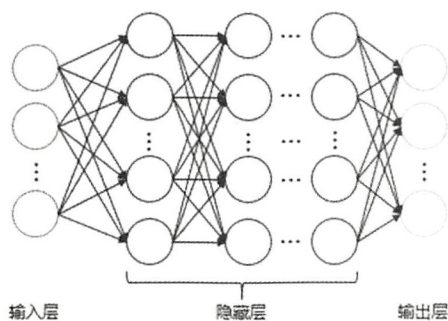


图 6-2

在 BP 神经网络中，一共可以分为三层：第一层为输入层，代表模型数据的输入；第二层为隐藏层，这一层为数据的处理层，在这一层中可以包含许多层；第三层为输出层，用来表示经过隐藏层计算后数据模型的输出。在这三层中，每一层的神经元之间采用的都是全连接的方式，且每一条边都有自己的权重，每一层都会有一个相应的偏置量。

对于 BP 神经网络的训练，一般会包括四个阶段。

1. 正向传播（向前传播）

对于一个有 n 层的 BP 神经网络，其输入层为第 1 层，输出层为第 n 层，隐藏层为第 2 层到第 $n-1$ 层。那么，对于输出层的第 k 个神经元来讲，其输出为 O_k^n ，下标 k 表示第 k 个，上标 n 表示层数，其输入为 I_k^n ，第 n 层所包含的神经元个数可以用 $|n|$ 来表示，那么其输入/输出关系可以这样表示：

$$I_i^{n-1} \rightarrow \text{第 } n-1 \text{ 层} \rightarrow O_i^{n-1} \xrightarrow{w_{ij}^{n-1}} I_j^n \rightarrow \text{第 } n \text{ 层} \rightarrow O_i^n$$

将正向传播分解成三个小阶段，分别是网络的初始化阶段、隐藏层的输出阶段和输出层的输出阶段。在正向传播的过程中，输入的信息通过输入层、隐藏层流转至输出层，逐层传递和计算每一层神经元的实际输出值。下面先来了解下这三个小阶段。

（1）网络的初始化阶段

在这个阶段中，假设输入层的节点个数为 n ，隐藏层的节点个数为 l ，输出层的

节点个数为 m ，我们可以定义输入层到隐藏层的权重为 W_{ij} ，隐藏层到输出层的权重为 W_{jk} ，输入层到隐藏层的偏置量为 a_j ，隐藏层到输出层的偏置量为 b_k ，学习速率为 η ，激活函数为 $g(x)$ ，以 Sigmoid 函数作为其激活函数，表达式为

$$g(x) = \frac{1}{1 + e^{-x}}$$

(2) 隐藏层的输出阶段

在隐藏层的输出阶段，首先需要将隐藏层中的权重进行求和，再利用激活函数 $g(x)$ 将其输出，其表达式为

$$H_j = g\left(\sum_{i=1}^n W_{ij}x_i + a_j\right)$$

(3) 输出层的输出阶段

在输出层的输出阶段，实际上就是将隐藏层所求得输出进行求和，再加上相应的偏置量，其表达式为

$$O_k = \sum_{j=1}^l H_j W_{jk} + b_k$$

2. 误差计算

经过正向传播的三个小阶段之后，可以求得最后的输出值。但是，实际的输出与期望的输出是有一定偏差的，所以，完成正向传播之后还要计算期望值与实际分类的误差，具体操作步骤如下：

首先定义误差向量，一般来讲，误差向量的值是由期望输出 Y_k 减去输出层的输出 O_k 得到的：

$$f_{\text{error}} = \frac{1}{2} \sum_{k=1}^m (Y_k - O_k)^2$$

3. 反向传播（向后传播）

在经过正向传播和误差计算之后，如果还没得到所期望的输出值，就需要把误差信号按照原来正向传播的路径原路返回，并且再对每个隐藏层的各个神经元的权重进行修改，使得误差信号趋向最小。

在反向传播的过程中，我们所需要的权重更新公式为

$$\begin{cases} W_{ij} = W_{ij} + \eta H_j (1 - H_j) x_i \sum_{k=1}^m W_{jk} e_k \\ W_{jk} = W_{jk} + \eta H_j e_k \end{cases}$$

该公式是根据误差反向传播的过程，使用梯度下降算法使误差函数达到最小值，公式的具体推导过程在本节中就不再阐述，有兴趣的读者可以参考相关资料。

在反向传播的过程中，主要包括以下两个步骤。

（1）隐藏层到输出层的权重更新

在计算隐藏层到输出层的权重更新时，首先要做的就是计算从隐藏层到输出层的梯度和微分，再利用所求得的梯度和微分更新隐藏层到输出层的权重，其微分形式可以写成：

$$\frac{\partial E}{\partial W_{jk}}$$

其中

$$E = \frac{1}{2} \sum_{k=1}^m (Y_k - O_k)^2$$

令

$$e_k = Y_k - O_k$$

则

$$E = \frac{1}{2} \sum_{k=1}^m e_k^2$$

则求得

$$\frac{\partial e}{\partial W_{jk}} = \frac{\partial e}{\partial y_i} \cdot \frac{\partial y_i}{\partial w_{h_o}} = -e_k H_j h_o$$

$$\frac{\partial E}{\partial W_{jk}} = \sum_{k=1}^m (Y_k - O_k) \left(-\frac{\partial O_k}{\partial W_{jk}} \right) = (Y_k - O_k) (-H_j) = -e_k H_j$$

上式为隐藏层到输出层的权重更新，则权重更新的公式可以表示为

$$W_{jk} = W_{jk} + \eta H_j e_k$$

(2) 输入层到隐藏层的权重更新

在计算输入层到隐藏层的权重更新时，首先要计算从输入层到隐藏层的梯度和微分，再利用所求得的梯度和微分更新输入层到隐藏层的权重，其微分形式可以写成如下表达式

$$\frac{\partial e}{\partial W_{ij}} = \frac{\partial e}{\partial H_i} \cdot \frac{\partial H_i}{\partial w_{ij}}$$

与隐藏层到输出层的计算类似，同样是将等号右侧的两个表达式分开计算，最后得到

$$\frac{\partial e}{\partial H_i} = W_{jk} e_k$$

$$\frac{\partial y_i}{\partial w_{ij}} = H_j (1 - H_j) x_i$$

则求得

$$\frac{\partial e}{\partial W_{ij}} = \frac{\partial e}{\partial H_i} \cdot \frac{\partial H_i}{\partial w_{ij}} = W_{jk} e_k H_j (1 - H_j) x_i$$

上式为输入层到隐藏层的权重更新，则权重更新的公式可以表示为

$$W_{ij} = W_{ij} + \eta H_j (1 - H_j) x_i W_{ij} e_k$$

4. 偏置更新

BP 神经网络的最后一步就是更新各层的权重，也就是偏置量，一般利用各层神经元的梯度和微分修正连接的权重。在反向传播的过程中，我们所需要的权重更新公式为

$$\begin{cases} a_j = a_j + \eta H_j (1 - H_j) W_{jk} e_k \\ b_k = b_k + \eta e_k \end{cases}$$

在偏置更新阶段，一般分为两个步骤：

(1) 隐藏层到输出层的偏置更新

在隐藏层到输出层的偏置更新的过程中，使用下列表达式求得偏置更新：

$$\frac{\partial E}{\partial b_k} = (Y_k - O_k) \left(-\frac{\partial O_k}{\partial b_k} \right) = -e_k$$

上式为隐藏层到输出层的偏置更新，则偏置更新的公式为

$$b_k = b_k + \eta e_k$$

(2) 输入层到隐藏层的偏置更新

在输入层到隐藏层的偏置更新的过程中，使用下列表达式求得偏置更新：

$$\frac{\partial E}{\partial a_j} = \frac{\partial E}{\partial h_j} \cdot \frac{\partial H_j}{\partial a_j}$$

将等号右侧的表达式拆成左右两个部分进行运算，求得：

$$\frac{\partial E}{\partial h_j} = (Y_1 - O_1) \left(-\frac{\partial O_1}{\partial H_j} \right) + \cdots + (Y_m - O_m) \left(-\frac{\partial O_m}{\partial H_j} \right) = -\sum_{k=1}^m W_{jk} e_k$$

$$\frac{\partial H_j}{\partial a_j} = \frac{\partial g(\sum_{i=1}^n W_{ij}x_i + a_j)}{\partial a_j} = H_j(1 - H_j)$$

最后求得偏置更新公式为

$$a_j = a_j + \eta H_j(1 - H_j) \sum_{k=1}^m W_{jk} e_k$$

至此，BP 神经网络的整套推导流程就已经完成。BP 神经网络算法是典型的前馈神经网络算法，根据整套推导过程，可以总结前馈神经网络的特点如下：

- (1) 前馈神经网络可以通过逐层信息传递得到最后的输出；
- (2) 前馈神经网络是沿着一条线一直计算，直到最后一层，求出计算结果的；
- (3) 前馈神经网络包含输入层、输出层和隐藏层，其目的是实现从输入到输出的映射；
- (4) 前馈神经网络一般分为多层，每个相邻层之间全连接，不存在同层连接和跨层连接的现象。

6.1.2 神经网络中的时序信息

在神经网络中，大部分的数据都是具有时序性的。所谓的时序性，是指随着时间的变化而发生变化且前后之间还有着紧密联系的一种特性。举个例子来说，人的脚步的交替是有一定规律性的，其规律性会随着时间的变化而发生变化。在进行匀速行走的时候，其脚步交替呈现周期性，当由走路突然变成了跑步时，之前的周期性就会随着时间的改变而发生变化。

如果不引入 RNN，而是采用传统的神经网络来记录走路这一形态的时序信息的话，则可以将“走路”这个循环的动作拆分成“迈腿（左）—站立—迈腿（右）—站立”这四个基本动作，而走路的这一套完整的过程实际上就是将这四个基本动作进行拼接完成的。简单地说，先记录了每一个动作单独的状态，并将其组合，最后将这一整套结果输出。传统神经网络的这种处理方式的特点就是简单、清晰，但是其缺点

也非常明显：

- 使用传统神经网络，需要将操作全部输入之后再一次性地输入神经网络中，这对于实时操作是非常困难的。
- 使用传统神经网络会将时间属性转换掉，这样的话，对于一些需要在时间序列进行操作的信息处理来讲是相当麻烦的。

为了解决传统神经网络在处理有时序信息的操作时所遇到的这些通病，我们通常会引入 RNN 来对其进行处理。

6.2 详解循环神经网络

6.1 节详细地介绍了前馈神经网络和神经网络中的时序信息处理，并一步步地推导出了 BP 神经网络。从 6.1 节可知，传统的神经网络并不能解决与时序信息相关的问题，当遇到时序问题的时候，传统的神经网络往往只会将其转换成一个空间模型进行处理，但这并不是我们想要的。接下来将详细地介绍 RNN 及其在时序信息方面的处理和应用。

我们都知道，深度学习技术之所以能够发展得这么迅速，最重要的原因就是它可以自动地总结和归纳已有的信息，预测和推断未来的信息和数据。RNN 作为深度学习技术的一种神经网络，在时序问题的预测方面表现得尤为突出，尤其对于自然语言的处理，更是起着不可替代的作用。

在日常生活中，无论对话还是向其他人讲事情都属于自然语言处理的范畴。例如，对于一道简单的填空题，“明天早上第一节课是英语课，所以明天早上我应该带一本（ ）书”。如果用正常人类的思维，那么很容易知道应该填入“英语”两个字。之所以很容易答对，是因为我们能够根据上下文的信息去推测空白处要填写的内容。但是对于计算机，要想准确地填对这个空，却是非常困难的。

对于传统计算机的识别方法来说，拿到一个语句之后，计算机首先会对这个语句进行分词，将语句分割成一个个的词块。例如：“明天早上第一节课是英语课”这句

话会被这样切割：“明天 早上 第一节 课 是 英语 课”。分词之后，再将每一个得到的结果进行标注，使之成为一个特定且唯一的词。最后，再进行句法和语义的分析，将得到的分析结果形成计算机所能认知的“知识”，并存储到知识库。当人们需要进行语义理解的时候，会在知识库进行搜索，然后去查询“一本”这两个字后面最可能出现的是什么词组，并将这个词组填入空缺的横线处。但是这种方式准确率不高，因为在计算机中有可能会搜索到“一本”这两个字的后面出现最多的是“故事书”这三个字。显然这个空缺处如果填写“故事书”这个词的话，一定是不符合大众的思维逻辑的，我们可以引入循环神经网络来解决这个问题。

循环神经网络与卷积神经网络最大的区别就是引入了类似“记忆”的概念。在 RNN 中，每个节点的输出不仅仅依赖其输入，还会依赖上一个节点中所存储的相关数据信息，也就是“记忆”，使上一个节点中的内容依然对本轮训练有影响。例如，使用循环神经网络填词时，计算机就会取上一句中的关键词，也就是“英语课”这个词进行分析，得出结论：后一句要填的空白处有很大的概率会跟“英语”这个词有关，因此就会在横线处填入“英语”二字。

下面通过循环神经网络的网络结构来分析一下循环神经网络的具体运作状态，如图 6-3 所示。

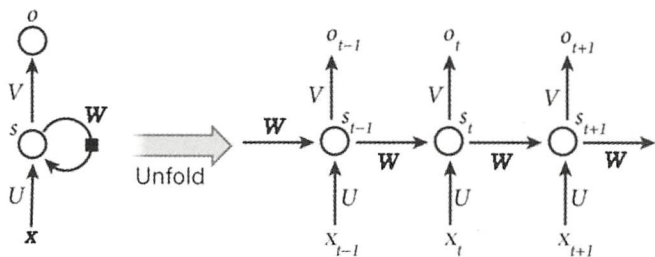


图 6-3

图 6-3 为一个典型的循环神经网络结构图，可以将右图看作左图的分解形态。首先由图 6-3 左侧部分可知，整个 RNN 的网络结构大体可分为三个部分，即输入层、隐藏层和输出层，这一点与卷积神经网络（CNN）很相似。但是值得注意的是，在 RNN 的隐藏层 s 部分会有一个环，这个环就是循环神经网络的特别之处，我们将其称

为循环递归， W 为循环递归的参数。

接下来看图 6-3 的右半部分，右半部分的箭头是从左侧开始指向右侧的，也就是说，RNN 是有一定的时间发展顺序的，从左到右可以看作一个 RNN 模型运算过程。在这个过程中， x 代表输入， t 代表输入的那个时刻， o 代表输出。当运算处于 x_t 时刻的时候，隐藏层的运算实际上所接收的是 2 个值：一个是来自本身时刻的输入，另一个是来自 x_{t-1} 时刻的输出。一般来讲，隐藏层的运算只接收当前时刻的输入。然而在这个模型中，我们也需要上一个时刻的输出值，说明 RNN 的运算过程实际上“记忆”了上一个时刻的输出值。从这个特性中很容易地看出，循环神经网络最擅长解决与时间序列相关的问题。

我们平时使用编程语言开发程序的时候，都会用到“循环”这个概念，通过循环来进行一些有规律的运算。循环神经网络之所以被这样命名，其原因自然也离不开循环这一基本特性。在循环神经网络中，可以将整个网络看作一个循环结构，而这个循环结构所做的事情就是在每一个时刻进行重复的运算，一般来讲，我们将这个被多次重复运算的结构称为循环体。继续观察图 6-3，可以发现无论在哪一个时刻，总会有一个参数 U 和参数 W 是不变的，在这里， U 代表输入层与隐藏层之间的连接参数，而不同时刻的隐藏层之间以参数矩阵 W 连接。在循环神经网络中，这些不变的参数为共享参数，而这些参数贯穿于循环神经网络的整个运算过程，这个过程称为参数共享。在循环神经网络中，参数共享能够很好地确保循环神经网络的每一步都在做相同的事情，极大地减少了在神经网络训练的过程中所需要的参数学习的工作量。

再反过来用循环神经网络分析前面的填空题。要预测空缺处词组，第一步就是要对整个语句进行分词，并为每一个分词建立一个词向量。可以令每一个词向量为循环神经网络中的一层，再根据循环神经网络的特点，在前面的语句中寻找单词，关注上一个单词的输出以及在本单词的输入，并利用前面的层中所存储的共享参数，使整个语句形成一个单词的序列，从而得出结果。

上面的例子讲解了循环神经网络在语言模型预测中的一个典型应用场景，其实循环神经网络除了预测下一个单词外还能做很多事情，例如翻译、语音识别、根据图像生成文字等。

6.3 RNN 的变种——双向 RNN

在前面的章节中重点讲解了循环神经网络 (RNN)，以及如何利用 RNN 进行单词预测，根据之前的知识可以知道，使用 RNN 进行预测，可以很好地利用之前所学习到的知识，以及在多个节点之间共享参数，从而达到预测的效果。但很多时候，我们的预测并不仅仅需要利用前面的内容，还要根据后面的内容来学习，换句话说，也就是结合上下文进行判断和预测。显然，传统的循环神经网络是无法满足这一需求的。为了解决利用下一个时刻的信息和上一个时刻的信息结合进行预测的需求，可以引入一种新的循环神经网络模式——双向循环神经网络 (bi-direction RNN)。顾名思义，双向循环神经网络相对于普通循环神经网络最大的区别就是其传播方向是双向的。在双向循环神经网络中，在任意的时刻 t 上，都会保证有两个隐藏层用来接收信息的传递，其中一个隐藏层用来接收从前往后的传递，另一个隐藏层用来接收从后往前的传递，然后将这两个隐藏层指向同一个输出层输出，以达到根据上下文进行预测的目的。在进行双向传递的时候，向前传播的方式和普通的 RNN 是一样的，而向后传播的方式可以理解为前面我们所讲解的 BP 神经网络的向后传播的方式。

下面来看一下双向循环网络在时间序列上的展开图，如图 6-4 所示。在每个时间序列点上都有 6 个权值，即从 w_1 到 w_6 ，并且这 6 个权值在每一个时间点上都是保持一致的，也就是说在双向循环神经网络中，也遵循着循环神经网络的基本思路，那就是参数共享。另外可以看到，每个时间点被分为四个层：输入层、向前传播层、向后传播层、输出层，并且向前传播层和向后传播层是相互独立的，两个层之间没有任何的信息交流。之所以这样做，是为了保证双向循环神经网络展开的图是一个非循环图。

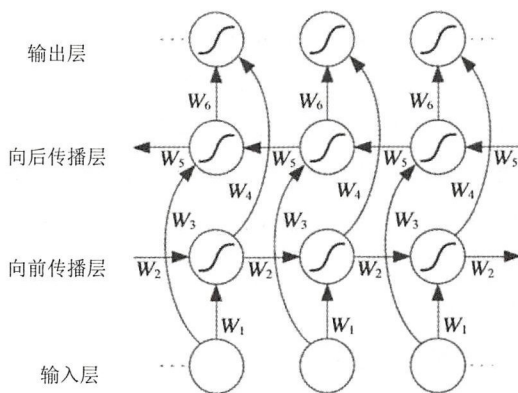


图 6-4

在 TensorFlow 中,已经提供了用来创建双向循环神经网络的接口,具体定义如下。

```
bidirectional_dynamic_rnn(
    cell_fw,
    cell_bw,
    inputs,
    sequence_length=None,
    initial_state_fw=None,
    initial_state_bw=None,
    dtype=None,
    parallel_iterations=None,
    swap_memory=False,
    time_major=False,
    scope=None
)
```

这个接口采用的原理就是独立的向前传播和向后传播方式,在使用此接口时,正向传播和反向传播的输入尺寸必须保持一致,在没有特殊规定的情况下,正向传播和反向传播的初始状态均默认为 0,并且不会返回任何的中间状态。

下面来解释一下这个接口的输入参数:

- **cell_fw**: RNNCell 的一个实例,用于向前传播。
- **cell_bw**: RNNCell 的一个实例,用于向后传播。

- **inputs**: RNN 的输入, 如果 `time_major==False` (默认), 则它必须是一个张量 `[batch_size,max_time,...]` 或者这些元素的嵌套元组; 如果 `time_major==True`, 则它必须是一个张量 `[max_time,batch_size, ...]` 或者这些元素的嵌套元组。
- **sequence_length**: batch 处理中每个序列的实际长度, 此参数是可选参数, 一般为一个 `int32/int64` 的向量, 大小为 `[batch_size]`。如果没有为其指定长度, 或没有提供这个参数, 则所有的 batch 条目均假定为一个完整的序列, 并且是从时间 0 到 `max_time` 的每一个序列。
- **initial_state_fw**: 向前传播的初始状态, 此参数是可选参数, 并且必须是一个张量 tensor, tensor 的形状为 `[batch_size,cell_fw.state_size]`, 如果 `cell_fw.state_size` 是一个元组类型的数据, 那么它应该是一个张量类型的元组, 并且在 `cell_fw.state_size` 中为其指定形状为 `[batch_size,s]`。
- **initial_state_bw**: 向后传播的初始状态, 与 `initial_state_fw` 相同, 但是必须使用 `cell_bw` 的属性进行操作。
- **dtype**: 初始状态和预期输出的数据类型, 是一个可选的值, 如果没有提供 `initial_states` 或者 RNN 状态具有异构的 `dtype`, 则这个值必填。
- **parallel_iterations**: 并行运行的迭代次数, 默认为 32 次。此参数的作用是用空间来换时间, 对于那些没有任何时间依赖性和可以并行运行的操作而言, 一般会使用它。当 `Values>>1` 时会使用更多的内存, 但是同时也会缩短运算的时间, 反之, 更小的值会使用更少的内存, 但是计算时间就会相对较长。
- **swap_memory**: 透明的交换在向前传播的过程中所产生的张量, 但是需要从 GPU 到 CPU 的运算支持。
- **time_major**: 张量形状的输入和输出。如果为 `True`, 则这些张量的形状必须为 `[max_time,batch_size,depth]`, 如果为 `False`, 则这些张量的形状必须为 `[batch_size,max_time,depth]`。使用 `time_major=True` 可以提高运算效率, 因为它可以避免 RNN 在计算开始和结束时的转制。但是由于大多数的 TensorFlow 数据是 `batch_major`, 所以默认情况下这个函数接收输入并以 `batch_major` 的形式进行输出。
- **scope**: 创建子图的变量范围, 默认为 `"bidirectional_rnn"`。

该接口以元组 (Tuple) 的形式返回数据, 格式为 `(outputs, output_states)`。其中, `outputs` 包含前向和后向; `rnn` 输出张量的元组格式为 `(output_fw,output_bw)`, 如果

`time_major==False` (默认), `output_fw` 将是张量类型: `[batch_size, max_time, cell_fw.output_size]`, 并且 `output_bw` 将是张量类型: `[batch_size, max_time, cell_bw.output_size]`。如果 `time_major==True`, 则 `output_fw` 将是张量类型: `[max_time, batch_size, cell_fw.output_size]`, 并且 `output_bw` 将是张量类型: `[max_time, batch_size, cell_bw.output_size]`。它返回一个元组而不是单个连接的张量, 与 `bidirectional_rnn` 不同。

在 TensorFlow 中, 如果要进行双向循环神经网络计算, 只需要调用 `bidirectional_dynamic_rnn` 方法即可。

6.4 One-Hot Encoding

无论是在深度学习领域, 还是在平常的数据处理和程序逻辑编写的过程中, 我们都会遇到针对类似枚举和某一类型的数据处理, 例如, 在做省份列表的时候, 我们会很习惯地将省份按照从 1~34 的序号转化成模型, 然后用相应的数值代入, 最后形成了如 `[0,1,2,3,4...33]` 这样的模型。虽然这样的转化会使程序运行更加高效, 但是对于深度学习的研究来讲, 这类数据是行不通的。因为在深度学习领域, 分类器会默认这些数据是连续的数据, 并且是有序的, 但是实际上并非如此, 这些编号仅仅是用来表示数据的一种形式, 真实数据是离散的, 因此, 这样的数据是不能直接放在深度学习算法中的。

One-Hot Encoding 正是为解决这类问题而产生的编码模式。One-Hot Encoding 又称为独热编码, 使用 N 位状态寄存器来对 N 个状态进行编码, 每个状态都有它独立的寄存器位, 并且在任意时候只有一位有效。例如在自然二进制编码状态为 001, 010, 011, 100, 101 中, 独热编码可表示为: 000010, 000100, 001000, 010000, 100000, 我们可以发现, 在上述独热编码中, 每个编码中只有 1 位是有效的, 也可以理解为每个编码实际上都是互斥的。

使用独热编码进行数据处理, 其优点就是能够处理非连续的数值特征, 并且会使这些特征变得稀疏, 从而也在一定程度上对特征进行了扩充。但是, 当特征的类型比较多时, 使用独热编码处理可能会使原本的数据变得过于稀疏, 从而增加了数据

后期处理的难度。

6.5 词向量和 word2vec

在进行自然语言处理的相关任务中，原始的数据是不能被计算机所识别和处理的，通常要先将这种自然语言转换成计算机能够理解的表示方式。计算机能够处理的无非就是一堆数学符号和相关字符，而词向量实际上就是将原始的输入进行抽象和处理，形成计算机能够识别的语言，再作为计算机的输入进行模型学习和训练。6.4 节我们所讲的独热编码实际上就是一种最简单的词向量。使用独热编码表示有一个缺点，就是不能很好地描述相似的词汇，使得两个意思相近或相同的词汇会变得非常独立，不利于后期使用。

而另一种词向量 Distributed Representation 则是我们在进行深度学习和自然语言处理中常用的表示方式，如果没有单独说明，那么词向量一般指的就是 Distributed Representation。

Distributed Representation 最早是由 Hinton 在 1986 年提出的，用来克服独热编码的缺点。试想一下，如果采用独热编码构建一个非常复杂的语言模型，则可能会需要建立一个具有非常庞大维度的模型，这个模型的维度可能是 10 000 维甚至 100 000 维。用这样的维度系数来表示一个语言模型，对于深度学习训练来讲无疑是灾难，但是如果此时用 Distributed Representation 来表示的话，效果则截然不同。

使用 Distributed Representation 表示是通过训练将一整段语句中的每一个词映射成一个固定长度的向量，而这个向量相对于独热编码来讲要短小精悍得多，然后将这些短小精悍的向量放在一起，形成一个向量的聚合空间。

一个原始的词只有经过一定的训练才能被表示成一个词向量。词训练的方法有很多种，在这里主要讲解 word2vec。

word2vec 是一个用来产生词向量的工具，其根本目的就是将单词转换成一个词向量。word2vec 的基本思想就是通过训练将每个词映射成 K 维的实数向量，并通过词



与词之间的距离来判断单词之间的相似程度，是基于 Distributed Representation 表示的。word2vec 模型中主要有两种模型，即 CBOW 和 Skip-Gram。从表面直观来讲，Skip-Gram 模型通过输入的单词来预测上下文，而 CBOW 则是通过上下文来预测空缺处单词的。下面分别讲解这两种模型。

6.5.1 CBOW 模型

CBOW 模型由输入层、映射层、输出层共同构成，其基本结构如图 6-5 所示。在这里，输入层一般使用上下文词语的词向量表示，而由于我们要做的就是训练词向量，所以这里的词向量理解为只是 CBOW 模型的一个参数而已。在训练开始阶段，我们会给定它一个随机值，然后随着训练的进行被不断地更新；映射层是对输入层所输入的向量做了一个求和，在这里仅仅是简单地对这些向量做了一个加法运算；输出层是用来输出最可能的值，一般来讲，输出层对应一棵二叉树，它是将语料中出现过的词作为叶子节点，以各个词在语料中出现的次数作为权值而构造出的哈夫曼树。

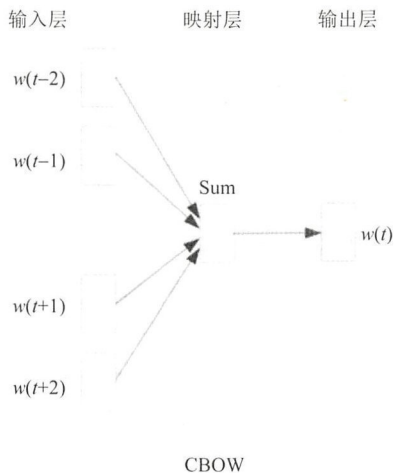


图 6-5

一般来讲，我们使用神经网络模型来做分类的时候，最常采用的方法就是使用 Softmax 进行回归，从而将每个分类输出的概率归一化。不可否认，用 Softmax 方法进行回归是一种简单明了的做法，在分类问题中效果也非常好。但是对于词向量分类

来讲，使用 Softmax 方法进行分类是一件非常头痛的事情，因为对于 Softmax 来讲，几十万个词汇量的语料实在太大了，在几十万个词汇上对每一个词汇都计算一遍输出概率并进行归一化处理，是一件非常费时费力的事情。

那么针对词向量，又应该如何处理呢？

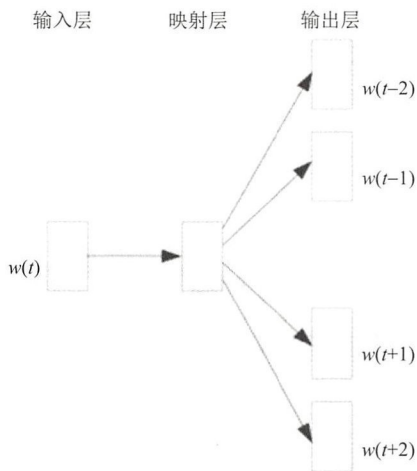
正如刚刚我们所讲的，利用 CBOW 模型实际上是一个二叉树结构，这也是与传统的神经网络模型相比的最大区别，这种二叉树结构应用到 word2vec 中被称为 Hierarchical Softmax，也完全可以将其想象成一个哈夫曼树的结构。Hierarchical Softmax 技术是 word2vec 中用于提高性能的关键技术，Hierarchical Softmax 不用为了获得概率分布而评估神经网络中的 W 个输出的节点，只需评估大约 $\log_2 W$ 个节点。Hierarchical Softmax 使用二叉树结构来表示词典里的所有词，使用每个词表示其中的一个叶子节点。在 Hierarchical Softmax 中，叶子节点的词没有直接输出的向量，而非叶子节点其实都有响应输出的向量。

6.5.2 Skip-Gram 模型

图 6-6 为一个 Skip-Gram 模型的结构图。从图 6-6 中可以很明显地看出，Skip-Gram 模型实际上和 CBOW 模型正好相反，也就是说，在 Skip-Gram 模型中，输入是一个特定的词向量，而输出是这个特定词向量所对应的上下文词向量。

Skip-Gram 模型与 CBOW 模型一样，也是由输入层、映射层和输出层构成的，不同的是，在 Skip-Gram 模型中，映射层包含当前样本的中心词 W 的词向量；而这个映射层实际上是一个恒映射的关系，也就是说，把词向量 $v(W)$ 映射到词向量 $v(W)$ ，在这里，映射层的目的仅仅是方便和 CBOW 模型做网络结构的对比，而这个输出层实际上和 CBOW 一样，也是生成了一个哈夫曼树。





Skip-Gram

图 6-6

6.6 梯度消失问题和梯度爆炸问题

6.1 节讲到了循环神经网络的基本特点，以及如何利用循环神经网络进行正向传播和反向传播，同时还提到了循环神经网络的一大特点，即能够利用前后记忆进行预测，这里的前后记忆包含了双向循环神经网络的基本知识。如此说来，利用循环神经网络及其双向循环神经网络（以下统称为循环神经网络），可以处理几乎所有与自然语言处理相关的问题，但是，事实上我们在处理更加复杂问题的时候利用这种基本的循环神经网络是远远不够的。

下面我们来看这么一段话：

“今天早上小明的妈妈教小明叠被子，首先要将它铺开，再从两个长边向中间叠，形成了一个长方形，最后把长方形的两个短边分别取三分之一处折叠，形成了一个更小的长方形，最后将长方形从中间对折，最终就完成了叠被子。”

这一段话比我们前面的例子所用的语言都要长，而且所需要得到的关键字“叠被子”实际上只在这段话的开头和末尾各出现过一次，如果我们使用经典的 RNN，就需要进行如下处理：

- (1) 利用分词工具（例如 jieba）将这段语句进行分词，使其形成一个个独立的单词；
- (2) 利用 word2vec 工具将所得到的分词转换成一个个的词向量；
- (3) 放到 RNN 中，进行向前和向后的反馈计算和学习，最终得到我们想要的结果。

理论上这是可行的，但实际上却会出现问题。因为我们需要的关键词“叠被子”在语句的开头出现，利用向后传播的理论，需要将这个词作为一个关键字，从 t_1 时刻一直传播到 t_n 时刻，在传播到最后一个时间点 t_n 时会得到一个误差；接下来利用向前传播算法，将这个误差反向传播回来，在进行反向传播的时候，会在途中经过的每一个时刻都乘以一个参数 W 。如果这个参数是一个小于 1 的数字，比如 0.8，那么这个误差在传递的过程中将会不断地减小，到最开始的时刻后，误差会变为 0.8 的 n 次方。当这个 n 足够大的时候，初始时刻的误差将会无限接近于 0，那么这个时候的误差似乎接近于消失的状态。但是这个结果肯定不是我们的预期结果，一般把这类问题叫作**梯度消失 (Vanishing Gradient)**；相反，如果每一个时刻的参数 W 是一个大于 1 的数字，那么最后求得的初始时刻的误差将会变得无限大，这个结果依然不是我们的预期结果，一般把这类问题叫作**梯度爆炸 (Exploding Gradient)**。随着网络层数的增多，梯度消失和梯度爆炸的问题就会表现得越来越明显。

6.6.1 梯度下降

目前神经网络模型的优化方法主要是梯度下降。我们使用梯度下降的方法进行误差的反向传播，不断地调整模型参数，以降低模型所产生的误差，使模型更好地实现从输入到输出的映射。目前因为各种因素，神经网络层数可以更深，神经元更多，相比以前得到了性能上较大的提升。



由于许多非线性层的作用，模型容量得到了较大的提高，使模型可以完成更加复杂的任务，模型很庞大，参数空间也非常复杂，我们使用的梯度下降算法是目前最有效的优化算法，但是这样深层的神经网络在误差反向传播过程中，很容易遭遇梯度消失和梯度爆炸的问题。

我们进行反向传播过程会把误差由输出层一层一层地往前面传播，这与神经网络的层数有一定的关系（如果是循环神经网络还会与时间步有关）。我们的误差是由链式法则一层一层地传播的，假设神经网络模型中的参数为 W ，则在链式法则中，需要多次乘以 W ，可以理解为 W 的 n 次方。假设 W 有特征值分解，则

$$W' = (V \text{diag}(\lambda) V^{-1})' = V' \text{diag}(\lambda)' V^{-1}$$

V 是由权重参数 W 矩阵的特征向量构成的矩阵， $\text{diag}(\lambda)$ 是由权重参数 W 矩阵的特征值 λ 构成的对角矩阵。

若 $\lambda > 1$ ， W 容易产生一个极大的数值，导致梯度爆炸。

若 $\lambda < 1$ ， W 容易更接近于 0，导致梯度消失。

梯度消失

每一次梯度更新的公式如下：

$$W = W - \alpha \frac{dJ(W)}{dW}$$

其中 W 为模型参数， α 表示学习率， $\frac{dJ(W)}{dW}$ 就是目标函数对参数 W 的导数。

如果产生梯度消失的问题，每一次的梯度更新， $\frac{dJ(W)}{dW}$ 就会无限接近于零，那么这样的梯度更新是没有意义的，这意味着已经无法进行学习了。

由链式法则可以知道，这样的问题经常出现在深层神经网络模型的较浅的层中。出现这个问题时，较浅的层往往还没有掌握最好的学习技巧和提取特征的能力，对于后续神经层及整个模型的效果都会产生较大的影响。如果只对后面的神经层进行训练，



前面较浅的层不再继续训练了，就不利于模型在参数空间中寻找最优解。

当循环神经网络中出现这样的问题时，可以理解为模型失去了对较早时刻的记忆，无法做到长期依赖，更加注重当前几步的信息，造成了时序信息的丢失。

梯度爆炸

深度神经网络模型很大，有许多非线性神经元，模型会呈现高度非线性，所以参数空间也会很复杂。我们可以将这么复杂的参数空间理解成一个地形非常复杂的地方，往往伴随着“悬崖”地形，在“悬崖”处的梯度是极大的，正因为这个极大的梯度才容易导致梯度爆炸的问题，如图 6-7 所示。

在进行梯度更新时，根据梯度消失部分说到的公式可以知道学习率乘以一个极大的梯度会导致参数更新的幅度非常大，离开了当前的区域，进入另外一个较远的区域，使之前更新的步骤都成了“无用功”，极大地影响了优化的性能。

在循环神经网络中，梯度爆炸出现得少一些，梯度消失出现得更多。梯度爆炸会更多地出现在深度前馈神经网络中。

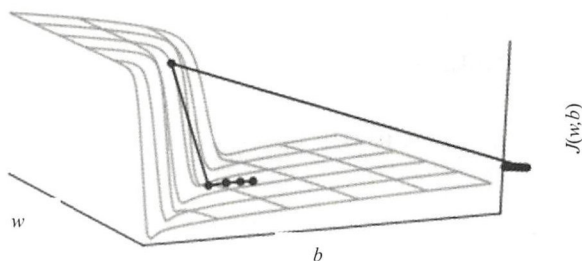


图 6-7

6.6.2 解决梯度消失和梯度爆炸问题的方法

选择合适的激活函数

在误差反向传播的过程中，需要对激活函数进行多次求导。此时，激活函数的导

数大小可以直接影响梯度下降的效果，过小容易产生梯度消失，过大容易产生梯度爆炸。如果激活函数的导数是 1，则这是最理想的情况，所以建议使用 ReLU 系列的激活函数，如 ReLU、ELU、Leaky ReLU，ReLU 函数图如图 6-8 左图所示，其导数图像如图 6-8 右图所示。

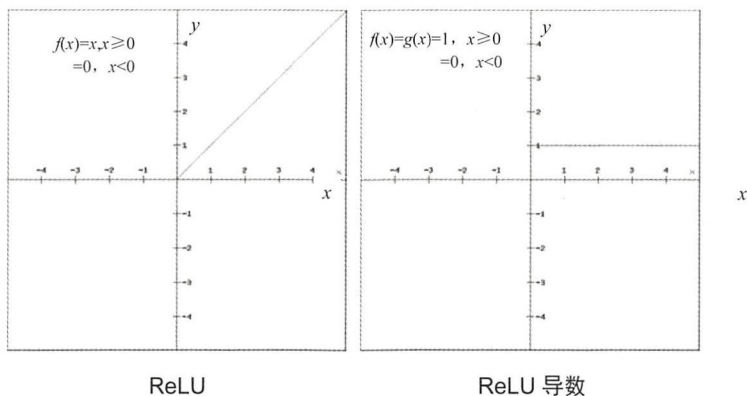


图 6-8

不建议大家使用 Sigmoid 和 Tanh 等激活函数，因为它们的导数在大部分区域都是非常小的，容易引发梯度消失的问题，如图 6-9 所示。

选择合适的参数初始化方法

在误差反向传播过程中，我们对当前层的输入求偏导数时，结果是就当前层的权重参数，将误差一层一层地传播下去，参数的大小也会影响梯度下降的效果。我们在进行参数初始化的时候是不可以把所有参数都初始化为零的。因为如果这样，参数就永远不会改变，是无法打破模型对称性的，从标准高斯分布中抽样，然后对它进行尺度变换（乘以 0.01），这样就得到了很多小的随机参数。

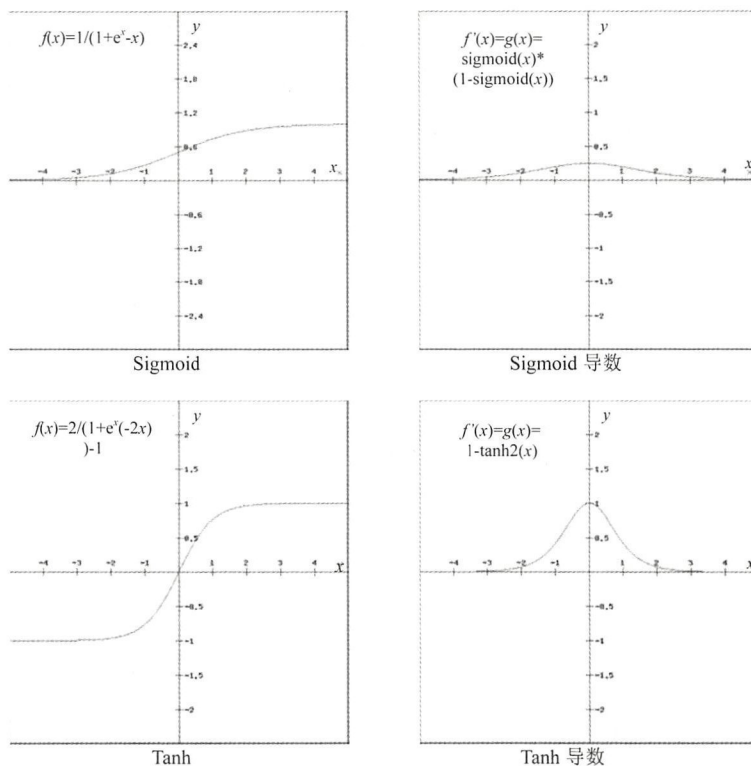


图 6-9

NumPy 公式如下：

$$W^{[L]} = \text{np.random.randn}(\text{shape}^{[L]}) \times 0.01$$

W^L 是第 L 层的权重参数， shape 是第 L 层权重参数矩阵的形状。

对于小工程，使用这样的参数可以打破参数对称的问题。但在结构深一点的网络可能容易发生梯度消失的问题，使得浅层网络得不到训练。当然，参数为 1 是最理想的情况，但显然这样是不合理的，我们初始化参数更多采用随机的方式，把参数初始化在一个合理的区间内且有合理的分布。

在神经网络前馈过程中，要求输入的加权和：

$$Z = W_1 \times X_1 + W_2 \times X_2 + \dots + W_n \times X_n$$

X 是输入, W 是该层的权重参数, Z 是输入的加权和。

在误差反向传播过程中, 对激活函数求导的结果就是 Z (输入的加权和), 我们很有必要对 Z 的大小进行一个有效的控制, 使其分布在一个合理的范围内。

因为假设 A, B, C 相互独立, 那么 $A+B+C$ 的方差大小为

$$\text{Var}(A+B+C) = \text{Var}(A) + \text{Var}(B) + \text{Var}(C)$$

假设参数矩阵 W 里元素之间是相互独立的, 可以知道随着 W 或者输入 X 维度的变大, $\text{Var}(Z)$ 的值很可能会很大, 使梯度不稳定, W 和 X 的维度不是我们可以随意改变的, 通过合适的权重参数初始化方法来约束 W 的分布以控制约束 $\text{Var}(Z)$ 的大小, 比如 HE Initialization 和 Xavier Initialization。

HE Initialization 是一种很适合使用 ReLU 系列激活函数的神经网络模型参数初始化方式。

NumPy 公式如下:

$$W^{[L]} = \text{np.random.random}(\text{shape}^{[L]}) \times \text{np.sqrt}\left(\frac{2}{n^{[L-1]}}\right)$$

$W^{[L]}$ 是第 L 层的权重参数, $\text{shape}^{[L]}$ 是第 L 层权重参数矩阵的形状, $n^{[L-1]}$ 是 $L-1$ 层的神经元数。

这种初始化方式将参数反差限定在一个合理的范围内, 能缓解梯度消失的问题。

如果将 Sigmoid 或者 Tanh 作为激活函数, 则可以考虑使用 Xavier Initialization。NumPy 公式如下:

$$W^{[L]} = \text{np.random.random}(\text{shape}^{[L]}) \times \text{np.sqrt}\left(\frac{1}{n^{[L-1]}}\right)$$

$W^{[L]}$ 是第 L 层的权重参数, $\text{shape}^{[L]}$ 是第 L 层权重参数矩阵的形状, $n^{[L-1]}$ 是 $L-1$ 层的神经元数。



使用权重参数正则化

使用权重参数正则化可以减少梯度爆炸发生的概率，常用的正则化方式就是 L1 或者 L2 正则化。对模型参数进行 L1 正则化时，参数会倾向于 0 和 1 的稀疏结构（假设参数为 Laplace 先验分布），对模型参数进行 L2 正则化时，参数会倾向于被“压缩”到一个很小的接近于 0 的数字（假设参数为标准高斯先验分布）。我们通过在目标函数中添加惩罚项达到这样的效果，在减小了模型复杂度的同时，也减小了发生梯度爆炸的概率，但是却增加了梯度消失的概率。

使用 BatchNormalization

BatchNormalization 目前已经在深度神经网络模型中得到了广泛的应用，主要通过规范化操作将输出信号 x 规范化到均值为 0、方差为 1，保证网络的稳定性，加快神经网络训练的速度，提高训练的稳定性，也可以缓解梯度爆炸和梯度消失的问题。

之前提过，在误差反向传播的过程中，经过每一层的梯度都会乘以该层的权重参数，举个简单的例子如下。

正向传播中： $f_3 = f_2(W^T X + b)$

那么反向传播中： $\frac{\partial f_3}{\partial X} = \frac{\partial f_3}{\partial f_2} \cdot \frac{\partial f_2}{\partial X}$ ，其中 $\frac{\partial f_2}{\partial X} = W$

反向传播式子中有 W 的存在，所以 W 的大小影响了梯度的消失和爆炸，BatchNormalization 就是通过对每一层的输出做规则模型和偏移的方法，用一定的规范化手段，把每层神经网络任意神经元这个输入值的分布控制在接近均值为 0、方差为 1 的分布，把偏离的分布强制拉回到一个比较标准的分布，这样使得激活输入值落在非线性函数对输入比较敏感的区域。这样输入的小变化就会导致损失函数较大的变化，使得梯度变大，避免梯度消失，而且梯度变大意味着学习收敛速度快，能大大加快训练速度，同时也能在一定程度上防止梯度爆炸的问题。从另一方面来说，机器学习中一种常用的数据预处理方法就是归一化，此时的归一化只针对输入数据，而 BatchNormalization 则将神经网络模型的每一层都归一化，从而让神经网络可以学习一种较为稳定的分布。



残差网络

残差网络 (Residual Network) 的提出在很大程度上解决了梯度消失的问题, 允许我们可以训练很深层的神经网络, 充分利用了神经网络里的每一个神经元, 模型更大更复杂, 也可以通过梯度下降的方式进行训练。现在的模型几乎都离不开残差网络的身影。

残差网络可以很轻松地构建至几百层, 上千层的网络, 而不用担心梯度消失过快, 原因就在于残差的捷径部分, 如图 6-10 所示。

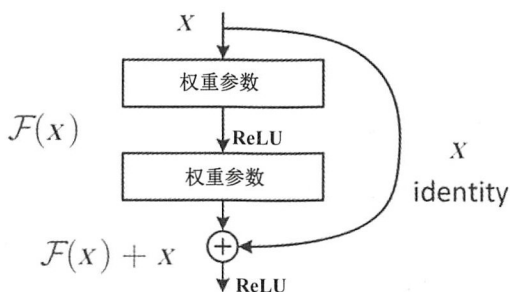


图 6-10

相比以前的神经网络模型, 残差网络中多了许多跨层的连接, 这样的结构在反向传播过程中有很大的好处, 见公式:

$$\frac{\partial \text{loss}}{\partial x_l} = \frac{\partial \text{loss}}{\partial x_L} \cdot \frac{\partial x_L}{\partial x_l} = \frac{\partial \text{loss}}{\partial x_L} \cdot \left(1 + \frac{\partial}{\partial x_L} \sum F(x_i, W_i) \right)$$

$\frac{\partial \text{loss}}{\partial x_L}$ 表示的损失函数到达 L 的梯度, 小括号中的 1 表明短路机制, 可以无损地反向传播梯度, 残差梯度则需要经过带有权重参数的层, 而非直接传递。残差梯度不会那么巧合地全等于 -1, 而且就算比较小, 短路机制的存在也不会导致梯度消失。所以残差结构可以被认为是解决梯度消失问题的最有效的、最重要的方法。

梯度裁剪

之前在讲解梯度爆炸产生的原因时, 提到了参数空间有很多“悬崖”地形, 导致

了梯度下降的困难，如图 6-11 所示，“悬崖”处的参数梯度是极大的，梯度下降时可以把参数抛出很远，使之前的努力都荒废了。我们解决这个问题的方法是进行梯度裁剪。梯度裁剪就是用来限制梯度大小的，若梯度大小超出了梯度范数的上界，则强制令梯度大小为梯度范数的上界的大小，来避免梯度过大的情况。在使用这样的方法进行梯度裁剪时，只是改变了这个梯度的大小，仍然保持了梯度的方向。

公式如下：

$$\text{if } \|g\| > \nu$$
$$g \leftarrow \frac{g\nu}{\|g\|}$$

其中 ν 是梯度范数的上界， g 用来更新参数的梯度。

如图 6-12 所示，我们要控制“悬崖”处梯度的大小，使用一个尽量小一点的梯度，避免穿越向上的曲面，使参数保持在一个合适的区域内。使用了梯度截断的梯度下降对“悬崖”处的反应更加温和，当参数更新到了“悬崖”截面处时，由于梯度大小受到了控制，不会那么容易被“抛出”到比较远的参数空间中去，导致“前功尽弃”。

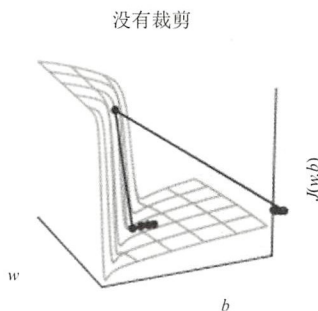


图 6-11

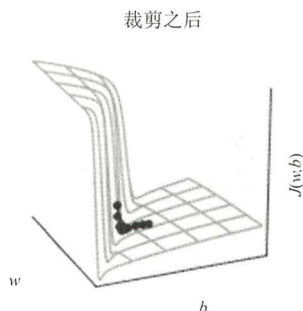


图 6-12

既然使用梯度裁剪的方式来处理梯度爆炸，同理，梯度消失可不可以使用梯度扩大的方式来解决呢？其实这个问题并没有那么简单，梯度过小有两种可能：一种是梯度消失，一种是到达局部最优或者鞍点。如果不能准确区分这两类情况，单纯扩张梯度有可能导致系统不收敛。而且梯度太小时，方向其实也是很难确定的，或者说很有可能是不准确的，不能随意地在某一个方向上放大梯度。

6.7 RNN 的变种——LSTM

刚刚分析梯度消失和梯度爆炸的问题，并且知道传统的循环神经网络在处理较为复杂的语句预测的时候似乎显得力不从心，为了解决这种长时间依赖的问题，研究人员在现有的 RNN 网络上提出了一种新的神经网络结构——LSTM。

LSTM (Long Short Term Memory Networks), 又被称为长短期记忆网络，实际上是 RNN 网络的一种特殊变种，正如其名字一样，不仅能对短期的文本进行预测和记忆，而且还能对离得比较长远的内容进行存储，来达到一个长期记忆的效果。可以说，LSTM 从最初的设计角度来讲就明确地避免了长期依赖的问题。

我们在本章开头的时候讲解了原始的 RNN 的网络结构，并且清楚地了解到其特点是循环，无论是在哪一个时刻，其权重值都是保持不变的，并且，原始的 RNN 神经网络只有一个输入值，这个输入值对于短期的输入是非常敏感的。那么 LSTM 网络又有什么不同呢？

图 6-13 是 LSTM 网络的每一个时间节点与原始 RNN 的每一个时间节点的输入状态的对比，通过对比不难发现，在 LSTM 网络中，我们增加了一个输入 C ，这个 C 实际上是另外一种状态，其目的就是用来存储长期的记忆，这个状态被称为单元状态 (Cell State)，我们将 LSTM 的这个时间节点按照时间顺序展开，得到如图 6-14 所示的结构。

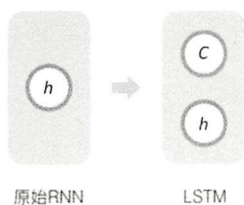


图 6-13

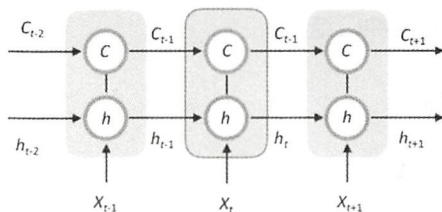


图 6-14

由图 6-14 可以看到，在 X_t 时刻一共接收到了 3 个值，这三个值分别是当前时刻的输入 X_t 、上一时刻的输出值 h_{t-1} 和上一时刻的单元状态 C_{t-1} 。在经过这一时刻的运

算之后，当前时刻 LSTM 的输出也是 2 个值，即当前时刻 LSTM 的输出值 h_t 和当前时刻的单元状态 C_t ，这就是在一个时间节点上做的操作。我们说 LSTM 实际上只是 RNN 的一个变种，那么这也说明 LSTM 同时也遵循着 RNN 的基本特性：循环。没错，LSTM 同样地也会将每一个时刻的操作进行复制，形成一个循环结构。

刚刚说到，在 LSTM 中有一个特殊的输入状态叫作单元状态 C ，用来保存和控制长期记忆，那么这个单元状态是如何工作的呢？

实际上这个单元状态可以看作由两个单元状态合并而成的，即上一个时刻的长期状态和当前时刻的即时状态。我们可以把这三个状态看成三个开关，而实现这三个开关的方法可以看作 3 个门——输入门（Input Gate）、输出门（Output Gate）、遗忘门（Forget Gate）。门是一种结构，其作用就是有选择性地让信息通过。在 LSTM 中，一般门是由一个 Sigmoid 神经网络层和一个元素相乘组成的。门实际上就是一层全连接层，其输入值是一个向量，通过前面的内容可以知道，因为 Sigmoid 函数的值域是 $(0,1)$ ，所以输出值则是一个 0 到 1 之间的实数向量，门的状态是半开半闭的。

当门的输出为 0 时，任何向量与其相乘都会得到 0 向量，在这种情况下，就相当于什么都不能通过；当门的输出为 1 时，任何向量与其相乘都会得到其向量本身，这就相当于什么都可以通过。在前面提到过的 3 个门中，遗忘门的主要作用就是决定了上一时刻的单元状态 c_{t-1} ，以及有多少保留到了当前的状态 c_t ；输入门的主要作用是决定了当前时刻的网络输入 x_t ，以及有多少保存到当前状态 c_t ；输出门决定了控制单元状态 c_t 有多少输出到了 LSTM 的当前输出值 h_t 。下面我们来简单说一下 RNN 和 LSTM 中的各种门结构。

LSTM

遗忘门（Forget Gate）:

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$$

输入门（Input Gate）:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$

输出门 (Output Gate):

$$O_t = \sigma(W_{of}x_t + W_{ho}h_{t-1} + b_o)$$

候选记忆细胞 (Candidate Cell):

$$\tilde{C}_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$$

记忆细胞 (Memory Cell):

$$C_t = i_t \times \tilde{C}_t + f_t \times C_{t-1}$$

输出隐藏层

$$h_t = \tanh(C_t) \times O_t$$

LSTM 的循环单元结构如图 6-15 所示。

GRU

更新门 (Update Gate):

$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z)$$

重置门 (Reset Gate):

$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r)$$

候选输出层 (Candidate Output Cell):

$$\tilde{h}_t = \tanh(W_{xh}x_t + W_{hh}r_t \times h_{t-1} + b_h)$$

输出层 (Output Cell):

$$h_t = (1 - z_t) \times h_{t-1} + z_t \times \tilde{h}_t$$

GRU 的循环单元结构如图 6-16 所示。

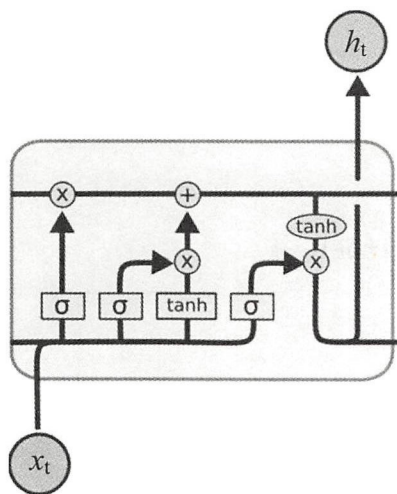


图 6-15

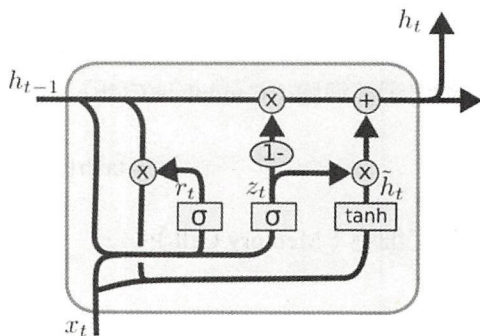


图 6-16

传统的 RNN 网络采用如下的公式累积过去的时序信息：

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b)$$

多个时间步就会有多层的嵌套，公式如下：

$$h_t = \tanh(W_{xh}x_t + W_{hh} \tanh(W_{xh}x_{t-1} + W_{hh} \tanh(W_{xh}x_{t-2} + W_{hh} \tanh(\dots)) + b) + b)$$

这个信息积累的过程如图 6-17 所示。

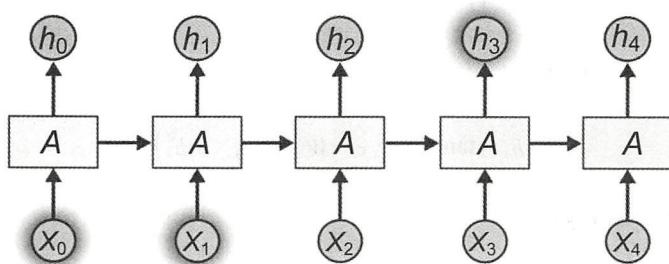


图 6-17

在 RNN 不断循环的过程中，通过非线性函数 Tanh 和权重参数 W 进行信息的累

积。在不断累积的过程中，我们对所有时间步的信息都“一视同仁”，并没有考虑到哪些信息对以后更重要，以进行一个有效的选择。同时这样由于非线性函数 \tanh 和权重参数 W 的原因，容易忽略掉以往时间步的信息，导致循环神经网络的记忆力很差。所以我们需要想出一种可以对以往时间步的信息和当前输入信息进行有效选择及有效累积的新的循环神经网络结构。因此，LSTM 和 GRU 这样的带有控制门的结构就应运而生。

LSTM 和 GRU 都加入了控制门，对当前输入的信息和以往的信息进行有效的选择和累积更新，LSTM 的门主要有三个：输入门、遗忘门和输出门。同时 LSTM 的隐藏层也不像传统循环神经网络一样，只用一个隐藏层，而是采用了三个不同的层来进行替代，分别为候选记忆细胞、记忆细胞和输出隐藏层。

输入门用来决定包含当前输入信息的候选记忆细胞有多少会被更新到记忆细胞中，对当前的输入信息进行一个有效的选择。遗忘门用来决定之前的记忆细胞有多少信息会被保留在当前的记忆细胞中。我们让神经网络在训练过程中逐渐学会选择哪些是重要的信息，哪些是不重要的。而输出门用来决定当前记忆细胞会有多少信息会被输出到当前隐藏层。记忆细胞的主要作用就是累积过往的信息，记忆细胞中的信息可能是对接下来某个时间步产生影响，不对当前产生影响。换句话说，就是记忆细胞中的有些信息对当前输出没有影响，我们需要通过输出门进行选择，将对输出层有用的信息从记忆细胞挑选到输出层中。LSTM 在进行过去时序信息累积和更新的过程中，使用如下的公式进行信息累积：

$$C_t = i_t \times \tilde{C}_t + f_t \times C_{t-1}$$

其中

$$\tilde{C}_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_{\tilde{c}})$$

这个候选记忆细胞已经包含了当前输入信息和之前输出信息，这是信息更新的主要部分，因为包含了新的信息。

所以

$$C_t = f_i \times C_{t-1} + i_t \times \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$$

如果是多个时间步就会有多层的嵌套，公式在这里就不展开了，可以自行推导。相对于传统的 RNN 结构用非线性结构进行信息积累的做法不同，LSTM 使用了乘法和加法运算的线性方法对信息进行积累和更新，我们对以往信息的选择没必要再经过非线性函数 Tanh 和权重参数 W 的作用了，可以直接通过遗忘门进行选择，有效地控制不同时间步信息对当前的影响，并不会逐渐丢失以往的信息，而使循环神经网络具有更好的记忆力。

GRU 是与 LSTM 功能很相像的一种循环神经网络结构，与 LSTM 又有细微不同的地方。GRU 也是使用了门控的结构，主要有两个门，分别是更新门和重置门，没有像传统循环神经网络一样用一个隐藏层，而是采用了两个隐藏层候选输出层和输出层。相比 LSTM，GRU 的结构更加简单，但其实起到的作用基本是一样的。重置门主要用来对之前输出层信息进行选择，选择有多少之前输出层的信息是可以进入用于信息更新的候选输出层中的，其实重置门是考虑了之前输出层对当前输入的影响，循环神经网络是需要考虑时序间相互关系的，当前输入与之前输出是有一定关系的，如在 NLP 任务中，之前输出层的输出是“不”字，当前的输入是“是”字，那它们是很明显相关的，组合起来就是词汇“不是”，当前候选输出层就可以表达出不同的意思了。如果没什么关系，那么重置门也会选择。候选输出层应该包含这样的信息。而更新门相对于重置门会考虑更加全局的信息，就好比考虑词汇“不是”是否应该保留。作用有点像 LSTM 的输入门，用来选择是否保留当前的输入信息，是否用包含了当前输入信息的候选输出层来更新输出层。输出层更新的公式如下：

$$h_t = (1 - z_t) \times h_{t-1} + z_t \times \tilde{h}_t$$

此公式只用一个门就起到了 LSTM 中两个门（输入门和遗忘门）的作用，降低了结构的复杂度，并没有像 LSTM 一样用额外的记忆层来进行信息的累积。同样，在重置门和更新门的共同作用下，GRU 也可以起到和 LSTM 中输出门一样的作用，对当前输出没作用但对未来时序有作用的信息也可以被暂时过滤掉，在接下来的步骤中可以继续使用。

GRU 使用如下的公式进行信息累积：

$$h_t = (1 - z_t) \times h_{t-1} + z_t \times \tilde{h}_t$$

多个时间步就会有多次的嵌套，其公式可以自行推导，和 LSTM 一样使用了乘法和加法进行信息累积和更新。我们对以往信息的选择没必要再经过非线性函数 Tanh 和权重参数 W 的作用了，可以直接通过更新门进行信息选择，使循环神经网络具有更好的记忆力。

其实无论是 LSTM 的输入门、输出门、遗忘门，还是 GRU 的重置门和更新门，这些门的产生其实都需要控制依据的，以上说到五个门的控制依据都是 $Wx_t + Wh_{t-1}$ ，然后经过 Sigmoid 函数压缩到 0 和 1 之间，控制依据和当前的输入及之前的输出有关，当然也可以构建其他的控制依据。而 GRU 和 LSTM 通常就是如上的控制依据。

每个门的控制依据都有不同的参数 W ，GRU 中控制门少一个，中间隐藏层也少一个，所以 GRU 的结构更简单，模型参数也更加少，避免过拟合等问题。目前对于很多任务，GRU 和 LSTM 的效果相当，LSTM 在某一些复杂任务上表现得更好，我们可以根据不同情况做出选择。

LSTM 和 GRU 之所以可以解决传统 RNN 的梯度消失问题，其实循环神经网络的“记忆能力”大小与梯度下降的效果有直接的关系，主要得益于它们进行信息累积时采用的线性方法，在进行 BPTT 过程中，我们可以通过一条“high-way”实现线性的求导，将梯度顺利地反向传播回去，没必要再次经过非线性函数 Tanh 和参数矩阵 W 的作用，这样梯度消失的概率也小了很多。传统 RNN 在 BPTT 过程中要经过非线性函数 Tanh 和参数矩阵 W ，容易导致梯度消失。这种思想和残差结构很像，通过这种方式提高了循环神经网络的记忆力，解决长期依赖问题。

其实可以将循环神经网络中的 Tanh 激活函数换成 ReLU 系列函数，也能在一定程度上缓解梯度下降问题，但效果肯定没有“high-way”这种做法好。

举一个例子，在传统 RNN 中：

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b)$$

令 $u_t = W_{xh}x_t + W_{hh}h_{t-1} + b$ ，如果 RNN 在时间步 t 产生了误差，在 BPTT 过程中误

差从时间步 t 传回到时间步 1，则需要计算 $\frac{\partial \text{loss}}{\partial h_1}$ ，则在传统 RNN 中：

$$\frac{\partial \text{loss}}{\partial h_1} = \frac{\partial \text{loss}}{\partial h_t} \frac{\partial h_t}{\partial u_t} \frac{\partial u_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial u_{t-1}} \frac{\partial u_{t-1}}{\partial h_{t-2}} \dots \frac{\partial h_2}{\partial u_2} \frac{\partial u_2}{\partial h_1}$$

其中 $\frac{\partial h_t}{\partial u_t}$ 是对 \tanh 求偏导且 $\frac{\partial u_t}{\partial h_{t-1}} = W_{hh}$ 。

熟悉 Tanh 函数的读者知道 $\frac{\partial h_t}{\partial u_t}$ 很可能是很小的数值，反向传播多个时间步下来梯度会非常小，容易产生梯度消失的问题。

在 LSTM 中，BPTT 过程中：

$$\frac{\partial \text{loss}}{\partial C_1} = \frac{\partial \text{loss}}{\partial C_t} \frac{\partial C_t}{\partial C_{t-1}} \frac{\partial C_{t-1}}{\partial C_{t-2}} \dots \frac{\partial C_2}{\partial C_1}$$

C_t 是时间步 t 的记忆细胞，我们知道 $\frac{\partial C_t}{\partial C_{t-1}} = f_t$ ，如果 $f_t=1$ 则是最理想的情况。

LSTM 的每个循环单元的 high-way 如图 6-18 所示。

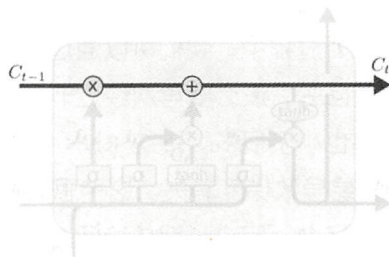


图 6-18

多次循环之后 LSTM 所构成的 high-way 如图 6-19 所示，图 6-19 中粗线就是多个时间步后所构成较长的 high-way，可以使 BPTT (Back Propagation Throng Time) 更加有效，通过遗忘门打开一条 “high-way”，将误差梯度通过时间反向传播回去，没必要经过非线性函数，从而避免了梯度消失问题，保持了完好的长期依赖功能和良好的记忆力。

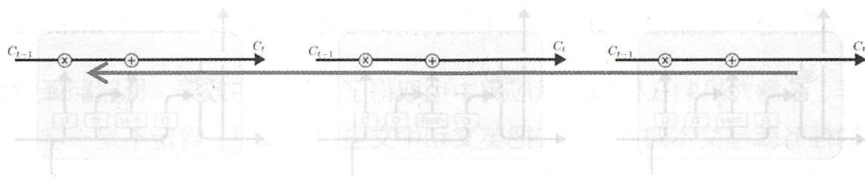


图 6-19

GRU 的分析方法和 LSTM 相同，GRU 则通过更新门打开了一条“high-way”，因为 $\frac{\partial h_t}{\partial h_{t-1}} = z_t$ ，同样解决了梯度消失问题，而且保持了良好的记忆力。

LSTM 和 GRU 可以通过“high-way”的方式很好地解决了传统循环神经网络的梯度消失问题，可以更好地累积过去的信息量。在经过“high-way”的反向传播途径中，LSTM 要经过遗忘门的作用，而遗忘门的数字在 0 到 1 之间，总会出现小于 1 的情况，如果我们输入的序列非常长的话，比如一篇一万字的文章，用 LSTM 或者 GRU 进行了编码之后，很可能就会丢失很早之前的部分信息，虽然这个问题没有传统 RNN 那么严重，但还是需要引起我们的注意。

机器翻译时往往使用的是 encode-decoder 的模型，先用循环神经网络“encoder”，再用循环神经网络“decoder”。但是遇到一些长句子的翻译时，效果就会变得很差了，在“decoder”的过程中，我们只会在“decoder”开始时把“encoder”的信息输入“decoder”循环神经网络，如果“encoder”的文本很长，那么根据 LSTM 的特性，可以知道最后“encoder”出来的信息肯定是不完整的，进而“decoder”无法得到完整的文本信息，所以翻译效果就很差。

注意力机制

目前解决这个问题的主要方法就是使用注意力机制 (Attention Mechanism)。注意力机制最早来源于计算机视觉的一项叫图像描述 (Image Captioning) 的任务中，在用 CNN 对图像进行了编码 (Encoder) 之后，再使用循环神经网络进行文本生成去描述这幅图像。在循环神经网络每生成一个词 (Word) 时，原图像中应该有某一片区域与这个词有着强烈的相关性，这跟人的视觉系统是一样的，总会先把注意力集中在某些细节，再考虑全局信息，那么用 LSTM 生成图像描述时，每一个词都可以捕获所对应

的图像区域信息。这就是计算机视觉用到的空间注意力机制（Spatial Attention）。

同样，注意力机制在机器翻译应用中也取得了很不错的效果，机器翻译中有一项很重要的任务就是文本对齐，例如把英文和中文进行翻译时，肯定不是一字一字对应过来解决问题的，每个中文词汇很有可能与其中好几个英文词汇的信息关联，当然也有没关联的词，我们要做的就是用注意力机制自动地挑选在翻译某个词时对原句子不同词汇的关联度。这些都是翻译需要解决的问题，在采用注意力机制捕捉之前“encoder”过程中的每个时间步的信息，然后输入“decoder”的每一个时间步中，这样“decoder”过程中每个时间步就可以得到更多的信息，同时用权重参数控制“encoder”时每个时间步的信息比重，再进入“decoder”的不同时间步中，也起到了选择的作用。

举一个例子。在进行文本翻译时，有一段原文本，合理分词后的对应的时间步长度为 t ，使用 LSTM 原文本进行“encoder”，每个时间步的输出隐藏层组成以下序列：

$$\langle h_1, h_2, h_3, h_4, \dots, h_{t-2}, h_{t-1}, h_t \rangle$$

在“decoder”的过程中再次使用 LSTM，假设根据“encoder”信息翻译成的目标句子长度为 t' ，每个时间步的输出隐藏层组成以下序列：

$$\langle H_1, H_2, H_3, H_4, \dots, H_{t'-2}, H_{t'-1}, H_{t'} \rangle$$

其中 h_t 是 t 步编码后得到的信息，在没有使用注意力机制之前，都是把 h_t 所包含的信息直接“decoder”，这样做的效果显然不好。用注意力机制时，要做的就是“decoder”生成输出隐藏层 H 时把之前“encoder”的每一个时间步的输出隐藏层 h 信息全部考虑进去。

在“decoder”的过程中采用如下的公式对“encoder”每个时间步的输出隐藏层信息进行选择：

$$c(t') = \sum_i \alpha'_i h_i$$

其中 α'_i 是权重参数， $c(t')$ 就是对“encoder”的每个时间步信息加权后的结果，其中：

$$\alpha_t^i = \frac{\exp(e_t^i)}{\sum_i \exp(e_t^i)}$$

且 $e_t^i = f(W_H H_{t-1} + W_h h_t + b)$ 。

上式得出， f 函数可以有多种形式，这里通常选择一个小型的神经网络，使用梯度下降去学习生成 e_t^i 的过程。

由“decoder”时前一时间步的输出隐藏层信息 H_{t-1} ，以及“encoder”过程的输出隐藏层信息 h_t ，共同决定 e_t^i 数值的大小。如果对门控循环神经网络熟悉的话，这里其实使用了跟控制依赖一样的思想，当然我们可以改变控制依赖的方式，由其他因素决定 e_t^i 的数值大小。

α_t^i 这个权重参数其实是一个 $t \times t$ 的矩阵，行对应的是目标句子的时间步，列对应的是元句子的时间步，每一行参数的和为 1，要用 Softmax 函数对每一行进行整理，反映了我们在“decoder”的每一个时间步对“encoder”每一个时间步的信息选择情况，权重和为 1，权重值越大说明对该“decoder”的时间步越重要。

自从注意力机制出现以来，各种各样的模型都引进了注意力机制。上面只是介绍了注意力机制之一，即 soft attention 方法。还有强化学习中的非参数的 hard attention，以及最近非常流行的 self attention 方法，当中思想基本都差不多，在这里不赘述，主要目的就是为了解决长期依赖和全局性信息选择问题。

6.8 写诗机器人

到此为止，我们已学习了 RNN 及其变体，并对分词技术和词向量技术有了一定的了解和掌握，在本节中，将会带领大家真真正正地实现一个写诗机器人。主要步骤如下。

- (1) 收集古诗词，并按照一定的格式整理到。
- (2) 将古诗词的题目和内容分离，然后进行数据清洗，其清洗的目的就是将那些

不好训练的样本去除，如含特殊符号、有不规则的字数，甚至一些生僻字。然后将清好的诗词的前后分别加上开始和结束符号，用来告诉 LSTM 这是诗词的开头和结尾。

- (3) 统计每个字出现的次数，并删除出现次数较少的生僻字。
- (4) 根据字出现的次数进行排序，建立字到 ID 的关系映射。
- (5) 构建 batch，每一个 batch 中所有的诗词都要补空格，直到达到诗的最长长度。
- (6) 搭建 LSTM 模型。
- (7) 训练模型。
- (8) 验证模型。

对古诗的处理

首先在互联网上收集古诗词，这里是按照“标题：内容”的格式整理的，例如：

幸秦始皇陵：眷言君失德，骊邑想秦馀。政烦方改篆，愚俗乃焚书。阿房久已灭，阁道遂成墟。欲厌东南气，翻伤掩鲍车。

立春日游苑迎春：神皋福地三秦邑，玉台金阙九仙家。寒光犹恋甘泉树，淑景偏临建始花。彩蝶黄莺未歌舞，梅香柳色已矜夸。迎春正启流霞席，暂嘱曦轮勿遽斜。

咏小山：近谷交萦蕊，遥峰对出莲。径细无全磴，松小未含烟。

赐萧瑀：疾风知劲草，板荡识诚臣。勇夫安识义，智者必怀仁。

这样做的目的是可以很方便地按照既定的规则切割诗词。

接下来便是整理古诗。通过冒号将古诗的标题和内容分离，将正文内容小于 10 个字符和大于 128 个字符的古诗全部过滤掉，还有将带有诸如“_”“《”“[”“(”等符号的古诗都过滤掉，再把古诗中的空格去掉，为每个古诗加上方括号“[]”作为开始和结束符号，最后将古诗的总数统计出来。具体代码如下：

```
poems = []
```

```

file = open(filename, "r")
for line in file: #every line is a poem
    #print(line)
    title, poem = line.strip().split(":") #get title and poem
    poem = poem.replace(' ', '')
    if '_' in poem or '《' in poem or '[' in poem or '(' in poem or '('
in poem:
        continue
    if len(poem) < 10 or len(poem) > 128: #filter poem
        continue
    poem = '[' + poem + ']' #add start and end signs
    poems.append(poem)
print("唐诗总数: %d"%len(poems))

```

接下来统计出诗词中全部单词的个数，具体代码如下：

```

allWords = {}
for poem in poems:
    for word in poem:
        if word not in allWords:
            allWords[word] = 1
        else:
            allWords[word] += 1

```

在得到了诗词中全部单词的个数之后，紧接着就要对这些词进行筛选，剔除掉那些不常见的单词，具体代码如下：

```

erase = []
for key in allWords:
    if allWords[key] < 2:
        erase.append(key)
for key in erase:
    del allWords[key]

```

使用 sort 函数根据字出现的次数进行排序，使用 lambda 函数及 dict 类型建立字到 ID 的关系映射。具体代码如下：

```

wordPairs = sorted(allWords.items(), key = lambda x: -x[1])
words, a = zip(*wordPairs)
words += (" ", )
wordToID = dict(zip(words, range(len(words))))

```

```
wordTOIDFun = lambda A: wordToID.get(A, len(words))
poemsVector = [[wordTOIDFun(word) for word in poem] for poem in poems]
```

处理诗词的最后一步就是为分词创建 batch，每一个 batch 中所有的诗词都要补空格，直到达到最长诗的长度，具体代码如下：

```
batchNum = (len(poemsVector) - 1)
X = []
Y = []
for i in range(batchNum):
    batch = poemsVector[i * batchSize: (i + 1) * batchSize]
    maxLength = max([len(vector) for vector in batch])
    temp = np.full((batchSize, maxLength), wordTOIDFun(" "), np.int32)
    for j in range(batchSize):
        temp[j, :len(batch[j])] = batch[j]
    X.append(temp)
    temp2 = np.copy(temp)
    temp2[:, :-1] = temp[:, 1:]
    Y.append(temp2)

return X, Y, len(words) + 1, wordToID, words
```

最后将所得到的序列、长度、单词所对应的 ID 及单词返回，用于构建训练模型，至此，对于古诗文本的处理部分就全部完成。

搭建 LSTM 模型

在对原始古诗文本进行整理和处理之后，接下来就要开始用处理好的数据来搭建 LSTM 模型了。

通过 TensorFlow 搭建 LSTM 模型，实际上和我们之前所讲的搭建 CNN 模型是大同小异的，其基本的原则都是定义模型、计算 loss、构建参数、训练、验证，得到最终的模型。

首先定义输入参数和输出参数：

```
gtX = tf.placeholder(tf.int32, shape=[batchSize, None])
gtY = tf.placeholder(tf.int32, shape=[batchSize, None])
```

接下来构建 LSTM 模型：


```
with tf.variable_scope("embedding"):
    embedding = tf.get_variable("embedding", [wordNum, hidden_units],
dtype = tf.float32)
    inputbatch = tf.nn.embedding_lookup(embedding, gtX)

    basicCell = tf.contrib.rnn.BasicLSTMCell(hidden_units, state_is_tuple =
True)
    stackCell = tf.contrib.rnn.MultiRNNCell([basicCell] * layers)
    initState = stackCell.zero_state(np.shape(gtX)[0], tf.float32)
    outputs, finalState = tf.nn.dynamic_rnn(stackCell, inputbatch,
initial_state = initState)
    outputs = tf.reshape(outputs, [-1, hidden_units])

    with tf.variable_scope("softmax"):
        w = tf.get_variable("w", [hidden_units, wordNum])
        b = tf.get_variable("b", [wordNum])
        logist = tf.matmul(outputs, w) + b

    probs = tf.nn.softmax(logist)
```

这段代码是构建 LSTM 模型的核心代码，让我们来好好地理解一下。

`tf.variable_scope()`方法的作用是实现变量共享，一般配合 `tf.get_variable()`方法使用，而 `tf.get_variable()`方法则是通过所给的名字来创建或返回一个变量。在本例中，实际上创建了一个名为 `embedding` 的变量。

`tf.nn.embedding_lookup()`方法用于在参数列表中执行并行查找，定义如下：

```
tf.nn.embedding_lookup(
    params,
    ids,
    partition_strategy='mod',
    name=None,
    validate_indices=True,
    max_norm=None
)
```

其中，`params` 参数一般被理解为分割一个比较大的“embedding”张量，其值一般是 `tf.get_variable()`的返回值。在本例中，也是这么使用的，我们所传入的“embedding”就是使用 `tf.get_variable()`方法获得的。

如果 `params` 的长度大于 1，那么就会根据 `partition_strategy` 指定的类型将 `ids` 中的每一个 `id` 元素分割在 `params` 元素之间。在所有的策略中，如果 `id` 的控件不能被均匀地分割的话，则每一个 `id` 将会按照 $(\max_id + 1) \% \text{len}(\text{params})$ 的结果进行分配。

`partition_strategy` 指定分区策略的字符串，只在 $\text{len}(\text{params}) > 1$ 的情况下才有效，目前支持 `mod` 和 `div` 两种类型，默认为 `mod`。此方法最后会返回一个与参数具有相同类型的张量。

本例使用 `inputbatch = tf.nn.embedding_lookup(embedding, gtX)` 将其结果赋值给一个名为 `inputbatch` 的参数，主要为了求得输入的张量。

接下来使用 `tf.contrib.rnn.BasicLSTMCell()` 创建一个基本的 LSTM 单元，并使用 `tf.contrib.rnn.MultiRNNCell()` 实例化 RNN，一般要向其传入 `basicCell` 的层数，例如这里使用 `stackCell = tf.contrib.rnn.MultiRNNCell([basicCell] * layers)` 实例化 RNN。

在构建 LSTM 网络的时候，一般使用 `stackCell.zero_state` 来获取 `initial_state`。

下一步，调用 `tf.nn.dynamic_rnn()` 创建一个由 `RNNCell` 指定的 RNN，并将返回的值赋给 `outputs` 和 `finalState` 两个参数。此时的 RNN 会完全动态地按照时序展开。

此时得到的 `outputs` 还需要进一步的处理，一般会调用 `tf.reshape()` 方法将张量 (tensor) 变换为阐述 `shape` 的形式，这里的 `shape` 一般为一个列表的形式。

最后使用 `Softmax` 函数完成 LSTM 基础模型的构建。

在构建完基础模型之后，接下来就要进行损失计算。本例使用 `tf.contrib.legacy_seq2seq.sequence_loss_by_example()` 来计算所有单词对应的 `label` 的加权交叉熵损失，然后使用 `tf.reduce_mean()` 来计算其代价，并将其赋值给变量 `cost`。

接着使用 `tf.trainable_variables()` 方法把刚刚创建的所有变量加入图中。

最后为其设置学习率、优化器，训练参数，并初始化一个全局的步数，用于计数。具体代码如下：

```
grads, a = tf.clip_by_global_norm(tf.gradients(cost, tvars), 5)
learningRate = learningRateBase
```

```
optimizer = tf.train.AdamOptimizer(learningRate)
trainOP = optimizer.apply_gradients(zip(grads, tvars))
globalStep = 0
```

上述的所有操作共同完成了一个 LSTM 网络的构建，接下来就要调用 `tf.Session()` 函数进行训练，并保存模型。代码如下：

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver = tf.train.Saver()
    if reload:
        checkPoint = tf.train.get_checkpoint_state(checkpointsPath)
        # if have checkPoint, restore checkPoint
        if checkPoint and checkPoint.model_checkpoint_path:
            saver.restore(sess, checkPoint.model_checkpoint_path)
            print("restored %s" % checkPoint.model_checkpoint_path)
        else:
            print("no checkpoint found!")

    for epoch in range(epochNum):
        if globalStep % learningRateDecreaseStep == 0: #learning rate
            decrease by epoch
            learningRate = learningRateBase * (0.95 ** epoch)
            epochSteps = len(X) # equal to batch
            for step, (x, y) in enumerate(zip(X, Y)):
                #print(x)
                #print(y)
                globalStep = epoch * epochSteps + step
                a, loss = sess.run([trainOP, cost], feed_dict = {gtX:x, gtY:y})
                print("epoch: %d steps:%d/%d loss:%3f" %
                    (epoch, step, epochSteps, loss))
            if globalStep%1000==0:
                print("save model")
                saver.save(sess, checkpointsPath +
                    "/poem", global_step=epoch)
```

由于篇幅有限，我们仅在此展示了如何进行文本的处理及其词向量的转化，以及如何构建 LSTM 和 RNN，并将其训练，得到一个新的模型。

第 7 章

TensorFlow 的可视化

到目前为止，我们已经可以训练出一些简单的 TensorFlow 模型，并利用这些模型进行预测，而针对目前我们所训练出来的模型是否可运用于生产环境当中这一问题，相信大家心里还是没底的。主要原因是我们并不确定训练出来的模型参数是否已经达到了最优的状态。一般来讲，一个模型如果想要运用于实际的生产环境中，还需要经过长时间的参数调整过程，尤其当训练的模型极其复杂的时候，调参工作可能需要持续数天或者数周的时间才能使模型达到一个相对最优的状态。

为了使模型达到最优状态，也为了方便开发人员进行参数调优和管理神经网络及模型，TensorFlow 提供了一个非常方便的可视化工具——TensorBoard。利用这个工具，可以很方便地将模型训练过程中的各种数据汇总起来，保存在一个自定义的路径与日志文件当中，也可以很方便地观察在模型运行过程中的各个指标随着时间变化而产生的变化趋势，并且在指定的 Web 端进行可视化展示。

7.1 TensorBoard 简介

TensorBoard 就是为了帮助开发者方便地理解、调试、优化 TensorFlow 程序而开发的。您可以用 TensorBoard 来展现 TensorFlow 图，绘制图像生成的定量指标图，以及显示附加数据（如其中传递的图像）。



一般来讲，TensorBoard 可以为开发者提供以下七种数据形式：

- (1) 标量 (Scalars) ;
- (2) 图像 (Images) ;
- (3) 音频 (Audio) ;
- (4) 计算图 (Graph) ;
- (5) 数据分布 (Distribution) ;
- (6) 直方图 (Histograms) ;
- (7) 嵌入向量 (Embeddings) 。

线性训练 MNIST 数据集所生成的 TensorBoard 界面如图 7-1 所示，该界面可以粗略地分成两个部分：上面的导航栏和下面的展示结果。导航栏用来针对不同的标量格式进行分类，在导航栏中可以很容易地切换不同维度的视图。而在不同维度的视图模式下，下面的展示结果也会随之发生相应的变化。

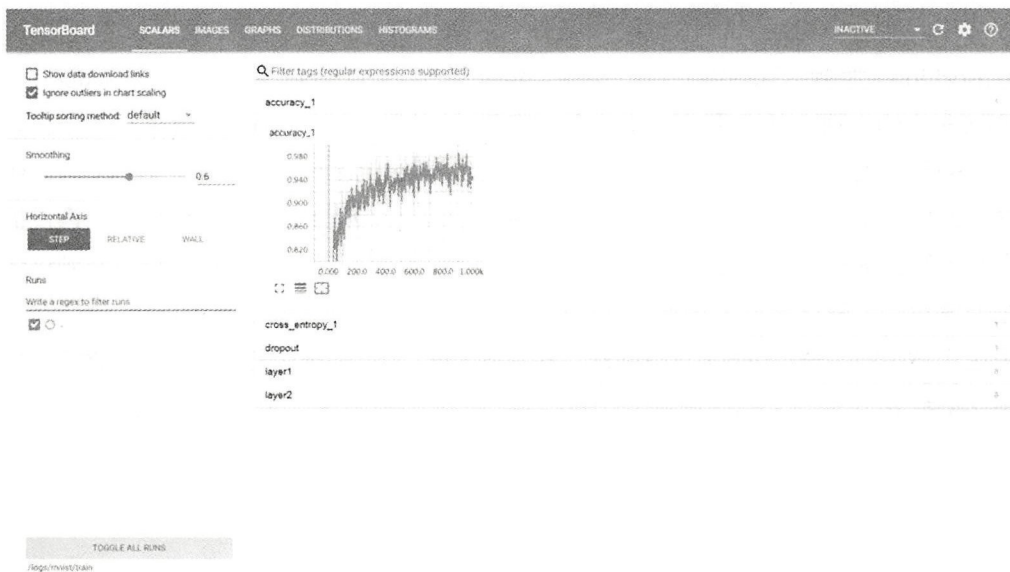


图 7-1



TensorFlow 的训练程序或测试程序与 TensorBoard 分别运行在不同的进程中。TensorBoard 实际上是通过读取 TensorFlow 产生的运行时事件文件来运行的。TensorFlow 的事件文件包含了运行 TensorFlow 时所生成的总结数据。一般来讲，当我们在自己的模型中引入 TensorBoard 时，首先要确定哪些数据需要被总结展示，再去考虑在 TensorBoard 中如何展示。比如，当使用卷积神经网络或线性结构来训练一个 MNIST 数据集时，我们希望能够记录学习速率和目标在训练过程中的变化情况，或者看到运行图的整体结构，以及在这些结构中某一部分的细节内容；再比如我们还希望显示一个特殊层中的激活分布或权重分布的情况等。这其中每一部分内容的记录，都需要使用 TensorFlow 中与之对应的函数进行调用。在 TensorFlow 中，所有与 TensorBoard 有关的数据记录都被包含在 `tf.summary` 这个类中。

在 TensorFlow 代码编写和训练的过程中，会存在着大量的变量及数以千计的节点信息，如果将这些节点信息一次性地全部展示在 TensorBoard 上，整个 TensorBoard 就会显得非常庞大且臃肿，所有的信息全部在一个层次上，甚至有些内容还无法看到或展示完全，总之给人的感觉可以用一个字来形容——乱。为了解决这个问题，TensorFlow 为我们提供了一个类似于命名空间的函数——`tf.name_scope()`。`tf.name_scope()` 函数允许我们为变量名划定范围，并且 TensorBoard 把该信息用于在图表中的节点上，为这个节点单独定义一个层级，层级的名称为 `tf.name_scope()` 函数所对应的值。在默认情况下，只显示顶层的名称和信息，内部的名称和信息可以通过折叠和展开的形式查看。图 7-2 是使用卷积神经网络训练的手写体程序，可以看到这个图非常复杂，有很多层级关系、参数和节点。实际上，这个图也是被折叠之后的图，把这个图中右下角的 `input` 节点展开，看一下 `input` 节点的前后对比，分别如图 7-3 和图 7-4 所示。





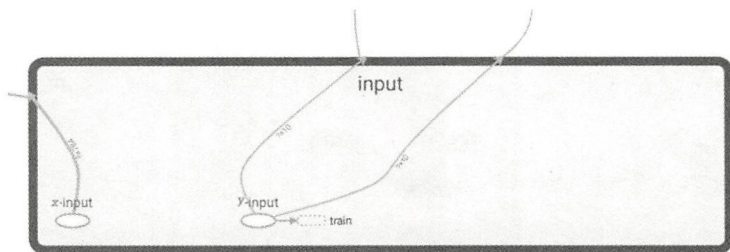


图 7-4

双击 input 节点，将 input 节点中的信息展开后，可以发现，在 input 节点中包含了两个小的节点：x-input 和 y-input。这两个小节点实际上对应着训练 MNIST 数据集的两个输入参数，而外面的一层 input 节点实际上就是使用 `tf.name_scope()` 函数所定义的一个层级。图 7-4 中所示的节点可用如下代码表示：

```
with tf.name_scope('input'):
    x = tf.placeholder(tf.float32, shape=[None, 784], name='x-input')
    y_ = tf.placeholder(tf.float32, shape=[None, 10], name='y-input')
```

结合这段代码再来仔细地看一下图 7-4，图中的 x-input 节点所连接的边上，有一个很小的参数为“ $? \times 784$ ”，这个参数实际上就是对应着上面代码 `x` 变量中的 `shape=[None, 784]`，而在 y-input 节点中，我们却发现了有两条边，这两条边所对应的参数为“ $? \times 10$ ”。在这里，“ $? \times 10$ ”是和代码中的 `shape=[None, 10]` 相对应的。而之所以有两条边，是因为 y-input 参数最终会在两个节点中用到：test 节点，以及在求 `cross_entropy` 时。

因此，`tf.name_scope()` 函数可以很方便地定义层级，降低阅读和使用 TensorBoard 的复杂度。

7.2 生成和使用 TensorBoard

7.1 节简单地介绍了 TensorBoard 的概念和作用，接下来详细地介绍如何生成和使用 TensorBoard 中的各种视图，并深入地解析 7.1 节提到的两个重要部分：`tf.summary` 类和 `tf.name_scope()` 函数。



`tf.summary` 类主要用于记录并产生各种事件文件的内容，`summary` 的操作相对于整个图来说是“外设”，所以 `summary` 需要调用 `run` 方法来执行。在 `summary` 操作中主要用到以下几个函数：

- **`tf.summary.scalar()`**：用来显示和记录标量数据，如记录准确率、损失值等。可以这样用它：

```
tf.summary.scalar('cross_entropy', cross_entropy)
```

- **`tf.summary.histogram()`**：用来显示和记录直方图数据，一般用来记录在训练过程中激活函数的分布、权重和偏置量等信息。可以这样用它：

```
tf.summary.histogram('activations', y_conv)
```

- **`tf.summary.text()`**：用来显示和记录文本输入类型的 `tensor`，一般可以将文本类型转换成 `tensor` 存入。可以这样用它：

```
text = "abcdefg123456"
summary_op = tf.summary.text('text', tf.convert_to_tensor(text))
```

- **`tf.summary.image()`**：用来显示和记录训练过程中的图像数据，可以这样用它：

```
x_shaped = tf.reshape(x, [-1, 28, 28, 1])
tf.summary.image('input', x_shaped, 10)
```

- **`tf.summary.audio()`**：用来显示和记录训练过程中的音频数据，可以这样用它：

```
def AddAudio(self, tag, wall_time=0,
step=0, encoded_audio_string=b'sndstr', content_type='audio/wav', sample_rate=44100, length_frames=22050):
    audio = tf.Summary.Audio(encoded_audio_string=encoded_audio_string,
                             content_type=content_type,
                             sample_rate=sample_rate,
                             length_frames=length_frames)
```

- **`tf.summary.merge_all()`**：将所有的 `summary` 信息合并，保存到磁盘中。
- **`tf.summary.FileWriter()`**：将 `summary` 和图信息保存到文件中，以供 `TensorBoard` 调用。

在使用 TensorFlow 训练模型的过程中，可以对所有需要观察的参数进行简单的分



类，通过分类可以知道这些变量和数据是使用上述函数中的哪一个或者哪几个来显示我们所需要信息的，将这些信息用 `tf.summary.merge_all()` 合并后，再用 `tf.summary.FileWriter()` 函数将其存入到相应的事件文件中，以供 TensorBoard 显示。

而 `tf.name_scope()` 函数的主要作用就是管理命名空间，以及命名空间下的参数。在 TensorFlow 中，管理命名空间的函数除 `tf.name_scope()` 函数外还有 `tf.variable_scope()` 函数，不过这二者实际上是存在着一定差别的，如下所示。

```
import tensorflow as tf
with tf.name_scope("my_scope"):
    a = tf.get_variable("value1", [1], dtype=tf.float32)
    b = tf.Variable(1, name="value2", dtype=tf.float32)
    c = tf.add(a, b)
print(a.name)
print(b.name)
print(c.name)
print("-----华丽的分割线-----")
with tf.variable_scope("my_scope"):
    a = tf.get_variable("value1", [1], dtype=tf.float32)
    b = tf.Variable(1, name="value2", dtype=tf.float32)
    c = tf.add(a, b)
print(a.name)
print(b.name)
print(c.name)
```

这段代码最后输出的结果如下：

```
value1:0
my_scope/value2:0
my_scope/Add:0
-----华丽的分割线-----
my_scope/value1:0
my_scope_1/value2:0
my_scope_1/Add:0
```

这段代码分割线的上半部分和下半部分除使用两个不同的命名空间方法外，其他都是一样的。再看一下输出，可以发现，在上半部分代码中，输出的 `a` 的 `name` 值为 `value1:0`，而在下半部分代码中输出 `a` 的值为 `my_scope/value1:0`，而其他输出信息不变。也就是说，`tf.name_scope()` 函数和 `tf.variable_scope()` 函数在这两部分代码中，针



对 `tf.get_variable()` 这个函数的处理逻辑不同, `tf.name_scope()` 针对 `tf.get_variable()` 的返回值是一个 `string` 类型的值, 而 `tf.variable_scope()` 针对 `tf.get_variable()` 的返回值实际上是一个变量, 换一句话说, 一个不带前缀, 而另一个则是带前缀的。

接下来将上面代码中的 `tf.variable_scope` 换成 `tf.name_scope`, 其他的不变, 再来运行它, 这个时候会发现, 代码报错了, 其报错信息如下:

```
ValueError: Variable value1 already exists, disallowed. Did you mean to set reuse=True or reuse=tf.AUTO_REUSE in VarScope?
```

很明显, 这个报错信息告诉我们, 变量 `value1` 已经存在了, 所以只有更名才能正常使用, 比如将 `value1` 换成 `value3`, 就会发现已经可以正常输出了, 输出内容如下:

```
value1:0
my_scope/value2:0
my_scope/Add:0
-----华丽的分割线-----
value3:0
my_scope_1/value2:0
my_scope_1/Add:0
```

通过这个小小的实验可以说明, `tf.variable_scope()` 允许变量同名, 包括 `tf.get_variable()` 得到的变量和 `tf.variable()` 的变量, 而 `tf.name_scope()` 则不允许让变量有相同的命名, 只是限于 `tf.variable()` 的变量。

另外, `tf.name_scope()` 函数在 TensorFlow 中的主要作用其实就是为操作张量的参数名称来划定范围, 并将信息传递到 TensorBoard 中用于可视化展示。

当我们把所有的参数和想要显示的数据都定义好了之后, 通常会用 `tf.summary.merge_all()` 函数来合并所有的参数, 并使用 `tf.summary.FileWriter()` 函数将其保存成文件, 例如: `tf.summary.FileWriter('./logs/tts/train', sess.graph)`。当事件文件被保存后, 可以在命令行中使用 TensorFlow 自带的 `tensorboard` 命令来运行 TensorBoard:

```
tensorboard --logdir=./mnist/logs/tts/train/
```

在这条命令中, `logdir` 是事件文件保存的路径。



到目前为止，我们已经了解了 `tf.summary` 类和 `tf.name_scope()` 函数。接下来看一个例子，该例子是对谷歌官方例子加以简单的改造而获得的。首先载入 MNIST 数据集，并定义一个函数来记录要汇总的数据信息。

```
import tensorflow as tf
import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)

sess = tf.InteractiveSession()

def variable_summaries(var):
    with tf.name_scope('summaries'):
        mean = tf.reduce_mean(var)
        tf.summary.scalar('mean', mean)
        with tf.name_scope('stddev'):
            stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
        tf.summary.scalar('stddev', stddev)
        tf.summary.scalar('max', tf.reduce_max(var))
        tf.summary.scalar('min', tf.reduce_min(var))
        tf.summary.histogram('histogram', var)
```

在这段代码中，我们首先使用 `tf.summary.scalar()` 函数记录了标量的信息，其中包括 `var` 的均值、标准差、最大值、最小值等，然后使用 `tf.summary.histogram()` 函数记录了变量 `var` 的直方图信息。

接下来定义权重和偏置量。

```
def weight_variable(para):

    initial = tf.truncated_normal(para, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(para):
    initial = tf.constant(0.1, shape=para)
    return tf.Variable(initial)
```

这里采用截断标准差来定义权重，并设置标准差 `stddev=0.1`。接下来就要定义卷积层并使用 `tf.name_scope()` 来定义其权重、偏置量等相关信息。

```
def nn_layer(in_data, in_dim, out_dim, layer_name, act=tf.nn.relu):
```




```

with tf.name_scope(layer_name):
    with tf.name_scope('weights'):
        weights = weight_variable([in_dim, out_dim])
        variable_summaries(weights)
    with tf.name_scope('biases'):
        biases = bias_variable([out_dim])
        variable_summaries(biases)
    with tf.name_scope('Wx_plus_b'):
        Wx_plus_b = tf.matmul(in_data, weights) + biases
        tf.summary.histogram('Wx_plus_b', Wx_plus_b)
    activations = act(Wx_plus_b, name='activation')
    tf.summary.histogram('activations', activations)
    return activations

```

首先定义了一个函数 `nn_layer`，以定义卷积层，并定义输入数据及其维度、输出数据及其维度，将激活函数默认为 ReLU 函数，接着使用 `tf.name_scope()` 函数定义三个命名空间，分别是输入信息权重信息 “weights”、偏置量信息 “biases” 和最后的求解信息 “Wx_plus_b”，并将权重和偏置量传入到前面定义的 `variable_summaries()` 函数中，最后使用 `tf.summary.histogram()` 将结果和激活函数传入到直方图中。

接下来定义输入数据和原始图像。输入数据直接使用 `tf.name_scope()` 函数定义命名空间即可。而对于原始图像，为了观察方便可将其传入到 `tf.summary.image()` 函数中，代码如下：

```

# 定义原始数据输入
with tf.name_scope('input'):
    x = tf.placeholder(tf.float32, [None, 784], name='x_input')
    y_ = tf.placeholder(tf.float32, [None, 10], name='x_input')

# 原始数据 reshape
with tf.name_scope('input_reshape'):
    x_shaped = tf.reshape(x, [-1, 28, 28, 1])
    tf.summary.image('input', x_shaped, 10)

```

接下来定义每一层的信息，以及计算交叉熵，代码如下：

```

# 创建第一层 layer
layer1 = nn_layer(x, 784, 500, 'layer1')
with tf.name_scope('dropout'): # layer1 后面 follow 一层 dropout
    keep_prob = tf.placeholder(tf.float32)

```




```

tf.summary.scalar('dropout_keep_probability', keep_prob)
dropped = tf.nn.dropout(layer1, keep_prob)

# 创建第二层 layer
y = nn_layer(dropped, 500, 10, 'layer2', act=tf.identity)

# 计算交叉熵 cross_entropy
with tf.name_scope('cross_entropy'):
    diff = tf.nn.softmax_cross_entropy_with_logits(logits=y, labels=y_) # diff
    with tf.name_scope('total'):
        cross_entropy = tf.reduce_mean(diff)
    tf.summary.scalar('cross_entropy', cross_entropy)

```

在这段代码中，使用 `tf.name_scope()` 函数定义了两个命名空间 “dropout” 和 “cross_entropy”，并将计算的交叉熵传入到标量统计当中。

最后定义训练信息、准确率等参数，并将准确率传入到标量的统计函数中，然后使用 `tf.summary.merge_all()` 来合并所有的 `summary` 信息，并使用 `tf.summary.FileWriter()` 函数将训练和测试的信息写入到文件中，以供 TensorBoard 使用。

```

with tf.name_scope('train'):
    train_step = tf.train.AdamOptimizer(0.001).minimize(cross_entropy)
with tf.name_scope('accuracy'):
    with tf.name_scope('correct_prediction'):
        correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
    with tf.name_scope('accuracy'):
        accuracy_op = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    tf.summary.scalar('accuracy', accuracy_op)

# 汇总所有 summary
merge_op = tf.summary.merge_all()
train_writer = tf.summary.FileWriter('./logs/mnist/train', sess.graph) # 保存 log 文件路径
test_writer = tf.summary.FileWriter('./logs/mnist/test', sess.graph) # 保存 log 文件路径

```

直到目前为止，我们将所有需要定义的部分都已经定义完成，接下来只需要进行正常的 MNIST 数据集训练即可，代码如下：

```

tf.global_variables_initializer().run() # 初始化所有变量

```



```

# 使用tf.train.Saver()创建模型的保存器
saver = tf.train.Saver()
for i in range(1000):
    if i % 10 == 0: # for test, 每10个step保存一次
        # 读入数据
        xs, ys = mnist.test.images, mnist.test.labels
        # 执行merge_op(数据汇总) 和 accuracy_op(准确率测试)
        summary, acc = sess.run([merge_op, accuracy_op], feed_dict={x: xs, y_: ys,
keep_prob: 1.0})
        print('Accuracy at step %s: %s' % (i, acc)) # 打印输出结果

    else: # for train
        if i == 99:
            run_options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
            run_metadata_op = tf.RunMetadata()
            xs, ys = mnist.train.next_batch(100)
            summary, _ = sess.run([merge_op, train_step], feed_dict={x: xs, y_: ys,
keep_prob: 0.6}, options=run_options, run_metadata=run_metadata_op)
            train_writer.add_run_metadata(run_metadata_op, 'step % 03d', i)
            train_writer.add_summary(summary, i)
            saver.save(sess, './logs/model.ckpt', i)
            print('Adding run metadata for', i)
        else:
            xs, ys = mnist.train.next_batch(100)
            # 执行merged_op和train_step, 记录summary
            summary, _ = sess.run([merge_op, train_step], feed_dict={x: xs, y_: ys,
keep_prob: 0.6})
            train_writer.add_summary(summary, i)
        # 训练完毕, 关闭writer
train_writer.close
test_writer.close

```

为了演示方便, 上面这段代码只训练了 1 000 轮, 训练之后, 会在./logs/mnist/train 目录下产生相应的事件文件, 如图 7-5 所示。

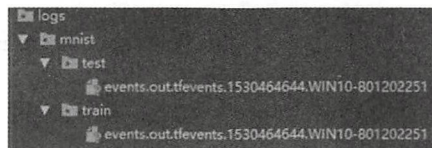


图 7-5



接下来要做的就是，使用 `tensorboard` 命令启动刚刚生成的事件文件，使用 `tensorboard --logdir=./mnist/logs/mnist/train/` 命令启动 TensorBoard。之后会在控制台输出如图 7-6 所示的信息。

```
(lite) C:\Users\Administrator\Desktop\tensorflow-mnist-master>tensorboard --logdir=./mnist/logs/mnist/train/  
TensorBoard 1.9.0 at http://WIN10-801202251:6006 (Press CTRL+C to quit)
```

图 7-6

根据提示在浏览器中使用上面的链接打开 TensorBoard（注意：建议使用 Chrome 浏览器进行访问），打开后界面如图 7-7 所示。

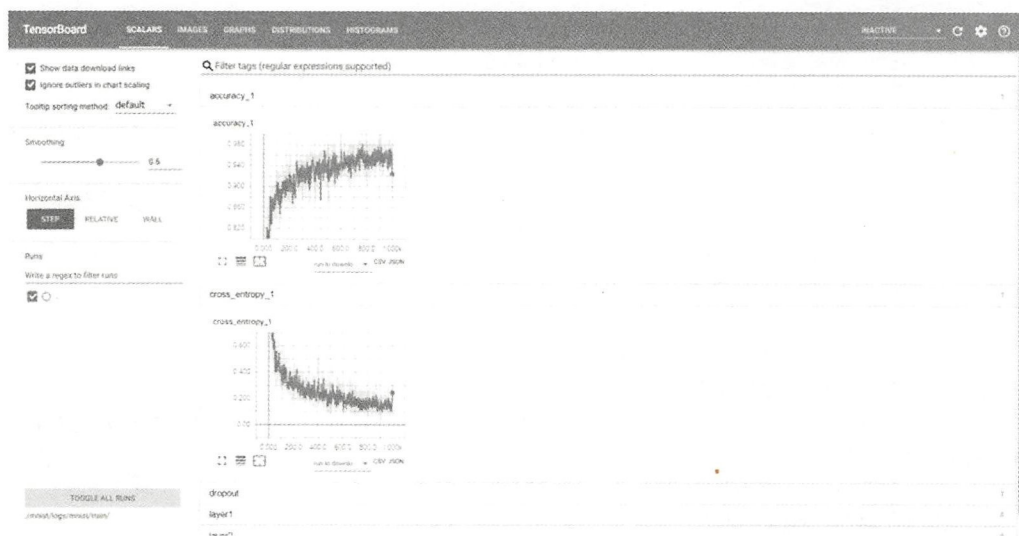


图 7-7

7.3 TensorBoard 的面板展示

至此，我们已经将训练的 MNIST 展现成了 TensorBoard 的形式，接下来就针对这个 TensorBoard 进行深入的解析。

先从导航栏开始，如图 7-8 所示。



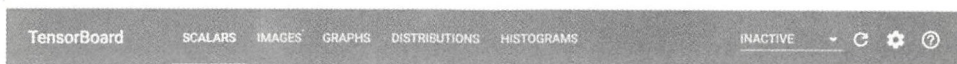


图 7-8

这个导航栏分为 3 个部分：最左侧是 TensorBoard 的 LOGO，中间为导航项目，右侧为其他功能区域。从中间的导航部分可以发现，一共有 5 个导航标题，分别为 SCALARS, IMAGES, GRAPHS, DISTRIBUTIONS, HISTOGRAMS，这 5 个导航标题实际上就是在代码中使用 `tf.summary` 类所产生的信息；导航栏右侧部分可以看到有一个下拉的小箭头，这个小箭头所具有的下拉选项指的是在 TensorBoard 中具备，但是实际上我们没有用到的一些信息，单击这个小箭头可以看到全部信息，如图 7-9 所示。

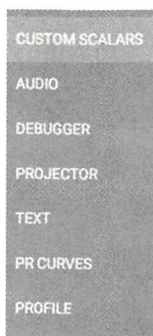


图 7-9

这些信息包括之前说过的 `tf.summary.audio()`、`tf.summary.text()` 等信息的展示，但是由于在这个项目中并没有相关的信息需要展示，所以在这里不做详细介绍。

接下来具体看看所用到的信息，这也是我们在日常生活中最常用的信息。

(1) SCALARS 面板

首先看 SCALARS 面板，如图 7-10 所示。

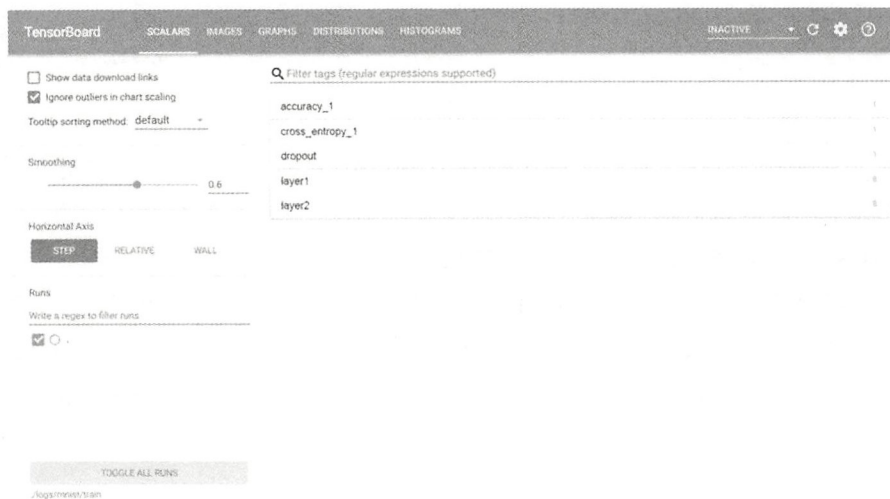


图 7-10

先将展开的图折叠起来，来讲一下大体的布局，SCALARS 面板的布局分为左和右两个部分，左边为一些常用的调节视图显示参数的面板，例如是否显示数据的链接、是否忽略外部链接、排序方式、平滑度、水平轴显示方式、运行一个参数等，如图 7-11 所示。

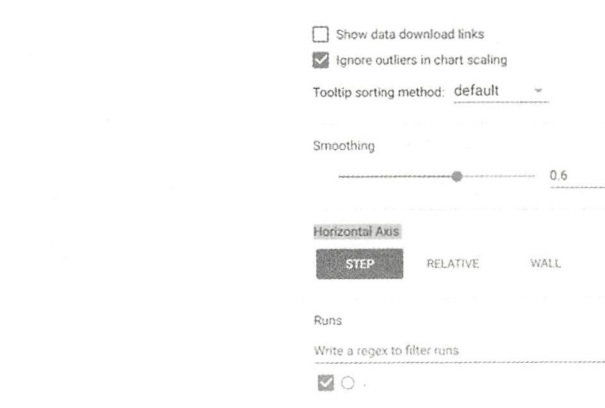


图 7-11

这里要着重讲平滑度（Smoothing）这个选项，这个选项实际上是为了调整右侧曲线的平滑程度。可以对比一下 Smoothing 在 0.6 和 0.9 下的曲线的变化，分别如图

7-12、图 7-13 所示。

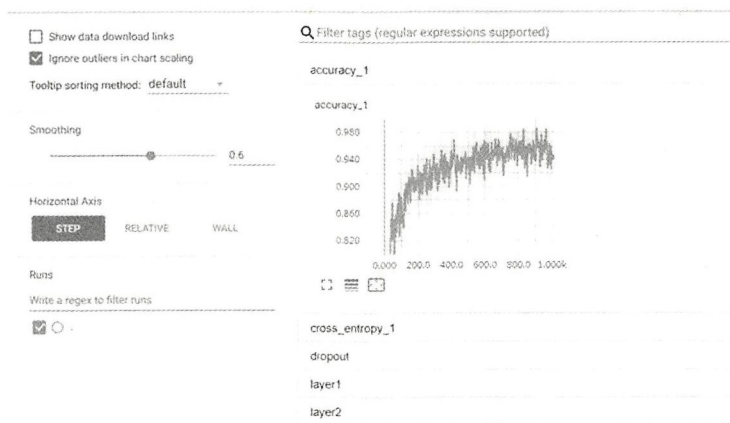


图 7-12

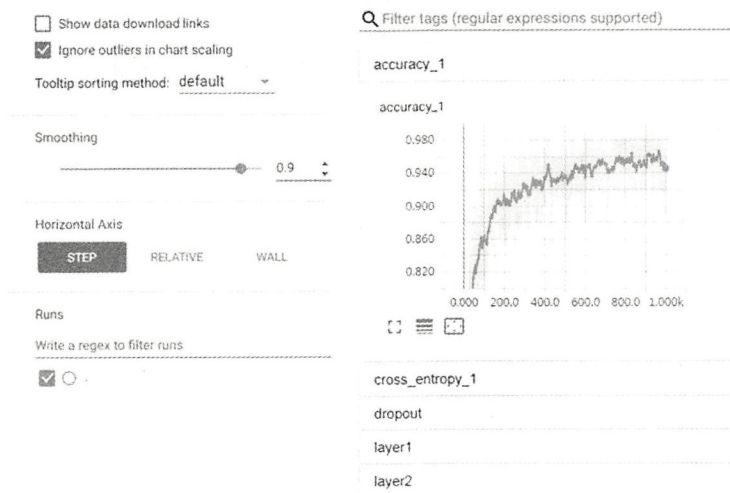


图 7-13

而 Horizontal Axis 参数主要用来调整右侧图像的显示方式，其中 STEP 是指按照步数进行显示的，例如图 7-13 中的右侧曲线，会以第 0 步到第 1 000 步的变化为横坐

标。RELATIVE 是指相对时间，也就是说从训练开始到训练结束所用的时间，这里的时间以小时为单位，如图 7-14 所示。

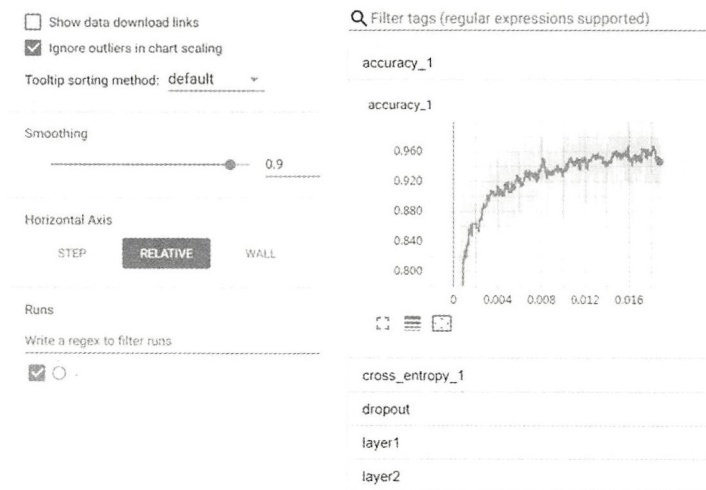


图 7-14

而 WALL 是指绝对时间，也就是说我们训练的时间，这个时间以系统时间为准，如图 7-15 所示。

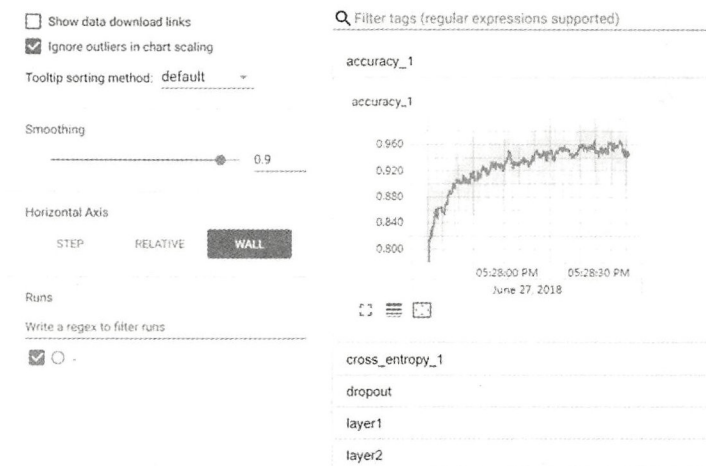


图 7-15

接下来再来看一下右侧的曲线图部分，这里显示的是准确率的监控曲线，如图 7-16 所示。

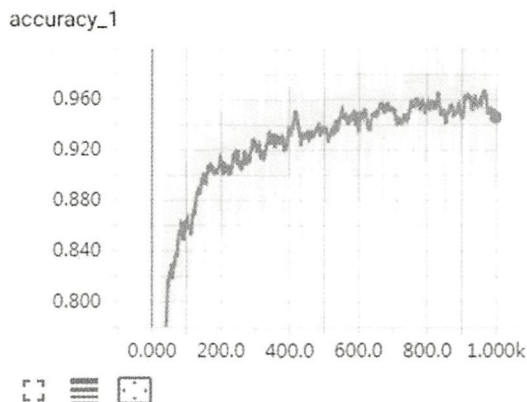


图 7-16

通过图 7-16 可以看到每步准确率的变化，从第 0 步开始，准确率为 0，但是随着训练步数的增加，准确率也在不断上升，到了第 400 步，准确率有一点小小的波动，不过总体还处于上升状态，到了第 950 步时，准确率达到了峰值状态，即约为 96%。通过这个曲线，我们可以很容易地观察到准确率的变化情况。通过图 7-16 还可以看到，在图的左下角有三个小按钮，分别是：放大曲线、是否对 y 轴数据进行对比、在拖动或放大图像之后恢复到原位置。

SCALAR 面板除了对准确率进行监控之外，还可以监控所需要监控的其他信息，如图 7-17 所示为监控 layer1 和 layer2 层的信息。

(2) IMAGES 面板

IMAGES 面板用来展示训练数据集和测试数据集的图像数据，通过 IMAGES 面板可以观察到在 TensorFlow 训练过程中最新的图像训练或测试使用的情况，IMAGES 面板如图 7-18 所示。

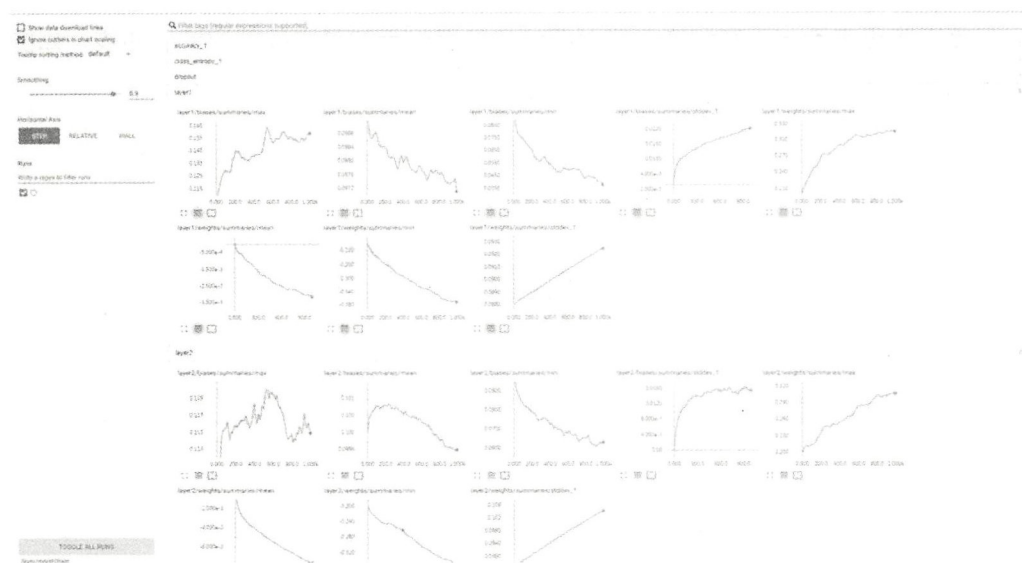


图 7-17

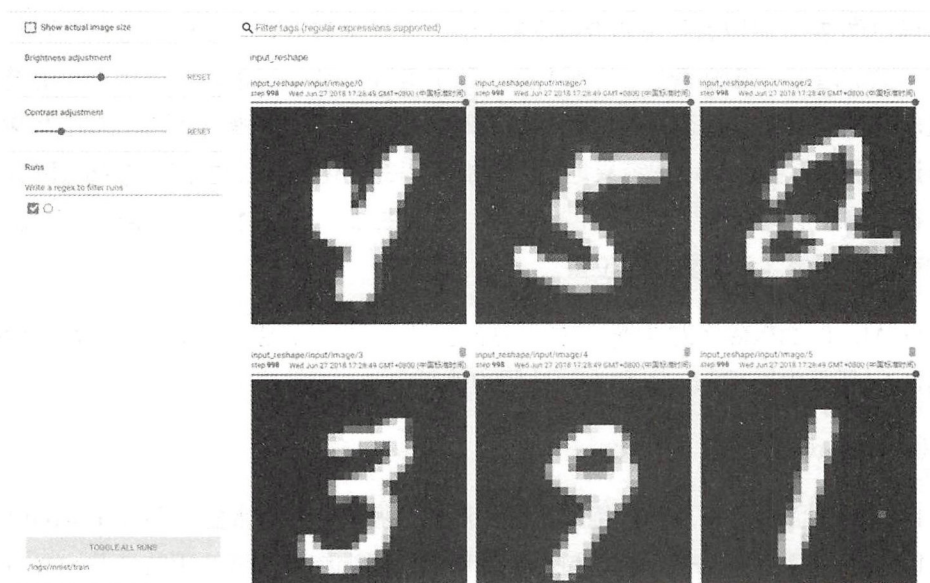


图 7-18

在 IMAGES 面板中同样包括左和右两个部分，左边为显示参数的面板，右边为作为输入的手写体图像。左边的参数面板如图 7-19 所示。

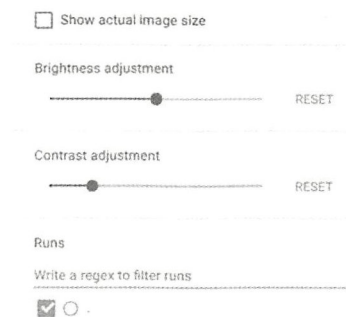


图 7-19

图 7-19 所示的参数面板主要有以下几个选项：

- (1) 展示图像的实际大小；
- (2) 亮度调整；
- (3) 对比度调整；
- (4) 运行其他参数。

为了显示方便，TensorBoard 一般会默认将图像的显示大小自适应地调整为适合屏幕的大小，而并不是实际大小，此时可勾选 Show actual image size，将图像显示调整为实际大小，比如 MNIST 数据集图像的实际大小为 28 像素 × 28 像素，如图 7-20 所示。这个大小实际上就是图像的原始大小，有的时候为了查看输入的图片是否统一可以采用实际大小显示。

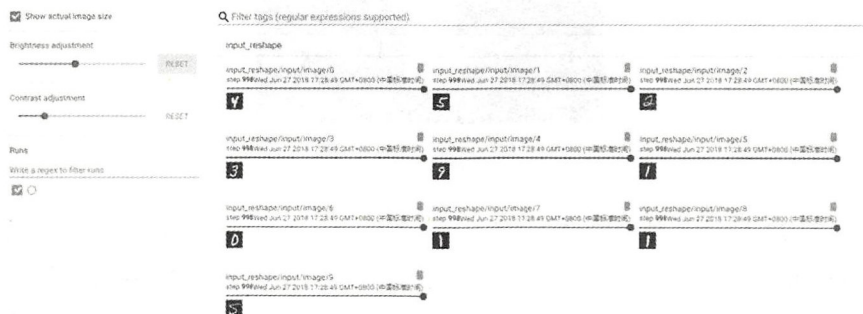


图 7-20

下面的 Brightness adjustment 和 Contrast adjustment 分别为亮度和对比度的调整，如果在图 7-20 的基础上，将图像的亮度和对比度调高，则会发现，右侧图像区域的亮度和对比度就会有明显的变化，如图 7-21 所示。

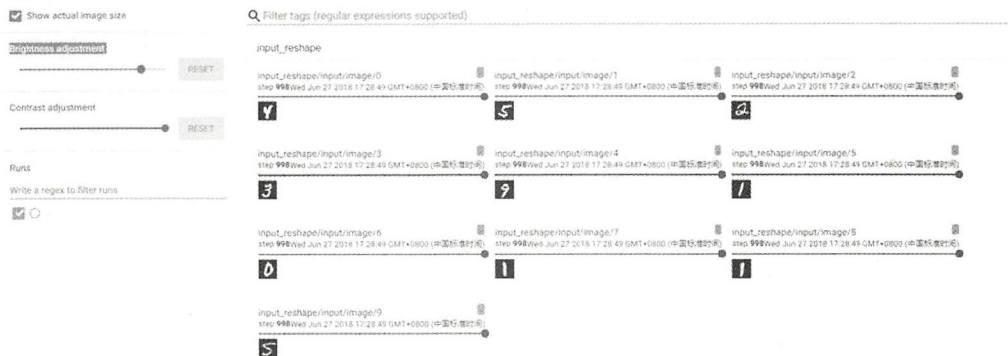


图 7-21

在右侧面板中，每一个图像的信息显示如图 7-22 所示。



图 7-22

如图 7-22 所示，整个图像信息包括两个部分，上半部分为图像的输入通道及训练的步数和时间，下半部分为当前所输入的图像，例如在图 7-22 中显示的是第 998 步所输入的图像为一个类似于数字 4 的图像。我们可以看到，在图像和信息之间，有

一个横向的拖动按钮，拖动此按钮可以调整当前通道不同的训练步数的图像输入信息，比如拖动到第 872 步，此时的训练输入图像类似于数字 6，如图 7-23 所示。

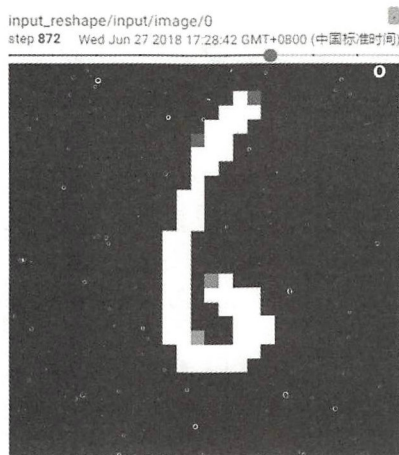


图 7-23

同时可以发现，随着训练步数的变化，训练的时间也在随之发生变化，也就是说在这里所显示的是当前训练步的时间。

(3) GRAPHS 面板

GRAPHS 面板用来展示整个训练所构建出来的图的结构信息，可以通过 GRAPHS 面板进行图的可视化，图中的信息就是在事件文件中所提取的 summary 操作的相关信息。

图 7-24 所示为 GRAPHS 面板所展示的信息。在左侧的参数面板中，可以调整在不同设备上（CPU 或者 GPU 等）所显示的颜色，以及看到各种图例的解析（左下角部分）。右侧的图信息面板显示的就是整个图的信息，实际上就是在代码中使用 `tf.name_scope()` 函数所记录的信息，以及使用 `summary` 所记录的相关信息，目前这个图中的每一个节点都是被折叠起来的，双击节点可将展开节点内部的结构和参数，如图 7-25 所示。

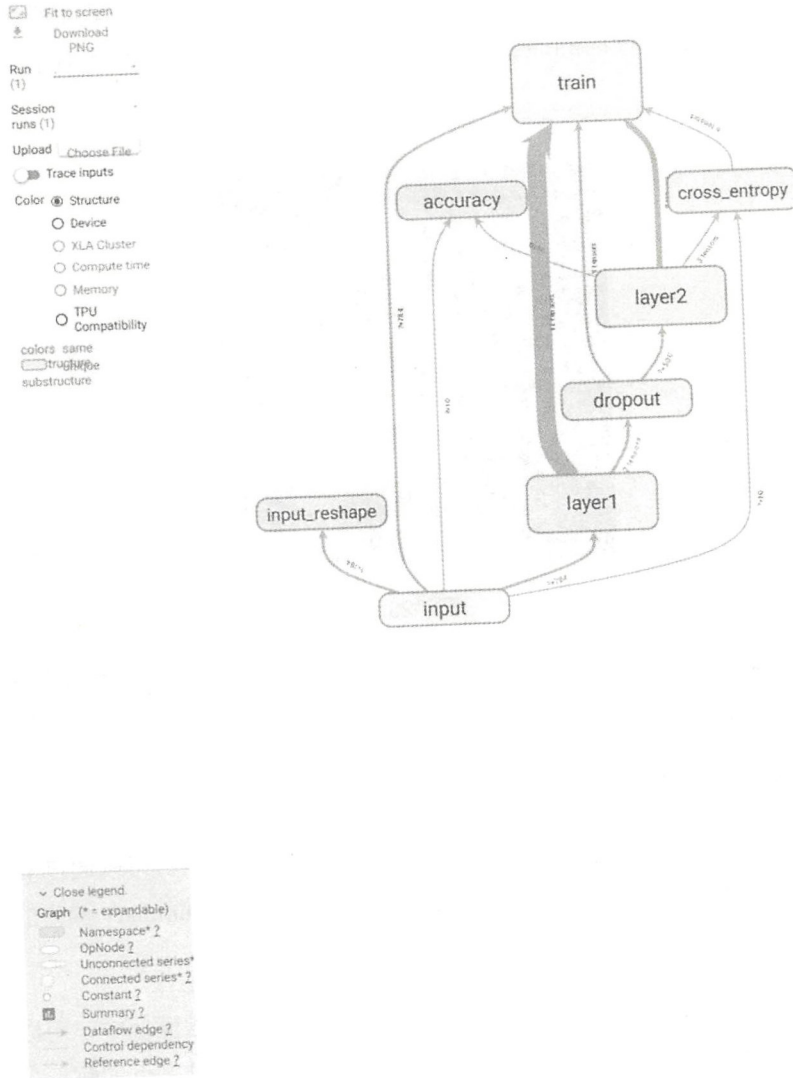


图 7-24

信息，还有向其中加入其他层使训练精度更高的情况。面对这些情况，单纯使用代码进行图的调整是非常费时费力的，尤其在改动图结构后。如果此时利用 GRAPHS 面板进行图结构的调整，确认位置后再对代码进行修改，就可以大大减少代码的出错率，提高整体训练效率。

(4) DISTRIBUTIONS 面板

通过 DISTRIBUTIONS 面板可以很容易观察到所训练的神经网络在不同的层中参数的取值分布情况，如图 7-27 所示。

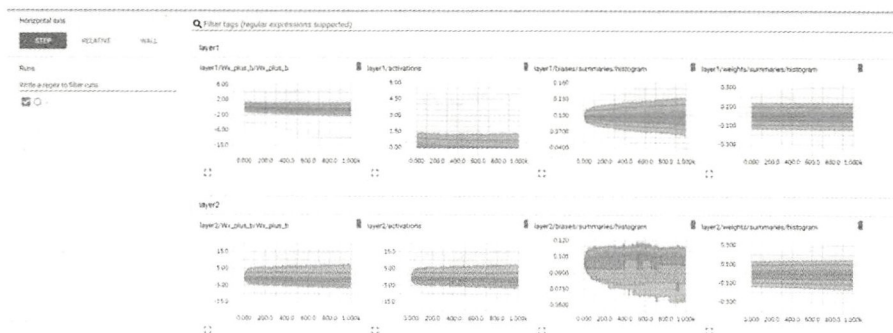


图 7-27

如图 7-27 所示，面板的左侧部分和 SCALARS 面板的左侧部分几乎是一模一样的，在此就不做详细说明。右侧部分则对 layer1 层和 layer2 层进行相应的信息监视，可以很清楚地看到在不同层中激活函数、权重、偏置量等信息的变化情况。

(5) HISTOGRAMS 面板

HISTOGRAMS 面板使用直方图来展示各个参数的分布情况。与 DISTRIBUTIONS 面板使用的是不同的维度，HISTOGRAMS 面板展示的是 TensorFlow 在不同时刻的参数分布。HISTOGRAMS 的信息取自 `tf.summary.histogram()` 函数所添加的信息，如图 7-28 所示。

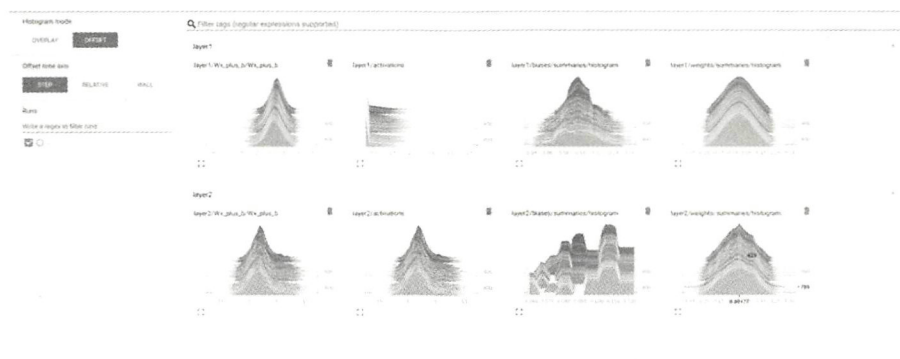


图 7-28

HISTOGRAMS 面板同样也包含了左侧和右侧两个部分，左侧参数信息部分如图 7-29 所示。



图 7-29

从图 7-29 中可以看到 HISTOGRAMS 提供两种模式，分别为 OVERLAY 模式和 OFFSET 模式。其中 OVERLAY 模式翻译成中文为覆盖模式，而 OFFSET 可以理解为偏移模式。在偏移模式下，可视化旋转 45°，各个直方图切片不再按时间展开，而是全部绘制在相同的 y 轴上。而在覆盖模式下，我们可以将其理解为多种直方图的重叠。

覆盖模式如图 7-30 所示。

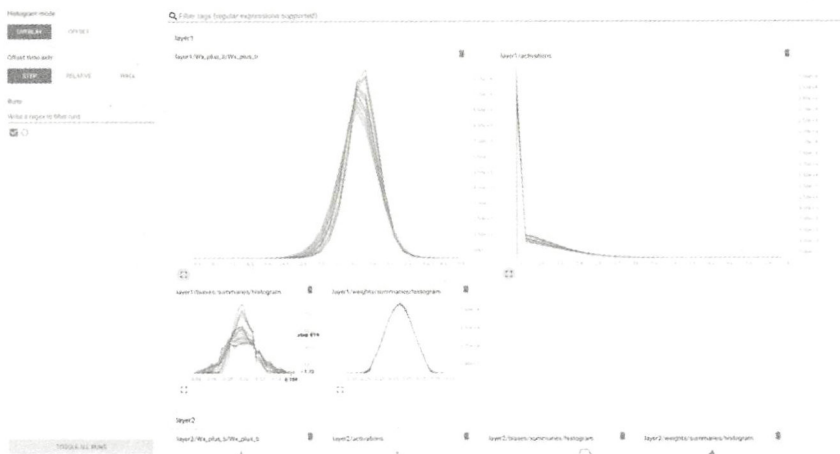


图 7-30

在覆盖模式下，每一个切片都是图表上单独的一个线条，y 轴显示每个存储区域内的项目数，颜色较深的线条表示较早的训练步，而颜色较浅的线条表示较晚的训练步。同样，您可以将鼠标悬停在图表上以查看其他信息。

偏移模式如图 7-31 所示。

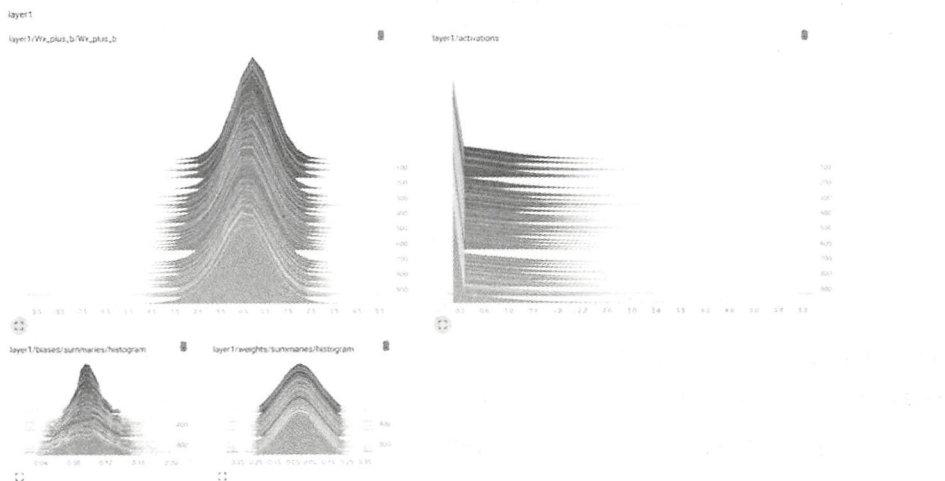


图 7-31

在偏移模式下, x 轴表示权重的值, y 轴表示训练的步数, 随着训练步数的增加, 颜色变得越来越浅。除左上角的正态分布外, 偏移模式同时还提供伽玛分布、均匀分布、泊松分布等分布形式, 通过各种分布形式观察 TensorFlow 的各个值的变化, 可帮助我们进行各种参数调优工作。

7.4 小结

通过本章内容, 我们知道如何使用 TensorBoard 观察参数及运行图。通过学习这些内容, 我们能够更好地对模型进行优化, 使模型变得更加健壮, 提高准确率。合理利用 TensorBoard 对模型调优有事半功倍的效果。

第 8 章

TensorFlow 中的数据操作

第 5 章详细地介绍了卷积神经网络,并了解了如何通过 TensorFlow 训练卷积神经网络,并将训练的结果加以验证,达到最终的训练目的。但是值得注意的是,我们之前所有训练的数据集都是使用别人处理好的数据集,且数据集的数据量都不是很大,因此训练会显得非常容易。而实际的场景中,我们可能需要面对各种各样的数据集,这些数据集有不同的尺寸和格式,如果直接利用其进行训练的话,显然是不可行的。因为大量的数据不仅消耗内存,而且使用起来非常不方便。为了解决这个问题,谷歌官方为我们提供了一套标准的格式,即 TFRecords。TFRecords 是一种能将图像数据和标签放在一起的二进制文件,这种文件可以很好地管理图像数据,并易于在 TensorFlow 中操作。

8.1 制作 TFRecords 数据集

在 TensorFlow 中,想要制作自己的数据集,首先需要准备数据,然后将准备的数据放在一个指定的目录下面,再使用 TensorFlow 制作成所需要的 TFRecords 格式的数据集,通过 `tf.train.Example Protocol Buffer` 存储。

这里提取猫狗大战中的前 300 张猫的图片 and 前 300 张狗的图片作为原数据制作自己的 TFRecords 格式的数据集,如图 8-1 和图 8-2 所示。



图 8-1



图 8-2

首先将猫和狗的图片分别存在 cat 和 dog 目录下，然后将它们统一放在了 D 盘的 testdata 目录中。

将上面的图片转换成 TFRecords 格式的文件，一般会经历以下步骤：

- (1) 定义 TFRecords 的文件名和存储路径；
- (2) 设定要分的类；
- (3) 将要训练的图片读入，并设定其尺寸；
- (4) 使用 Image 的 tobyte()方法生成二进制格式；
- (5) 使用 tf.train.Example 将图片和标签进行封装；
- (6) 将生成的 Example 序列化。

经历过以上步骤之后，就会在指定目录下多出一个 TFRecords 文件。下面就用刚刚的猫狗大战的图片使用 TensorFlow 生成 TFRecords 格式的文件。

代码如下：

```
import os
import tensorflow as tf
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np

cwd = "D:\\\\ testdata\\"
classes = {'cat', 'dog'} #预先自己定义类别
writer = tf.python_io.TFRecordWriter('dog_train.tfrecords') #输出成tfrecord文件

def _int64_feature(value):
    return tf.train.Feature(int64_list = tf.train.Int64List(value = [value]))

def _bytes_feature(value):
    return tf.train.Feature(bytes_list = tf.train.BytesList(value = [value]))

for index, name in enumerate(classes):
    class_path = cwd + name + '/'
    for img_name in os.listdir(class_path):
        img_path = class_path + img_name #每个图片的地址

        img = Image.open(img_path)
```

```
img = img.resize((208, 208))
img_raw = img.tobytes() #将图片转化为二进制格式
example = tf.train.Example(features = tf.train.Features(feature = {
    "label":
_int64_feature(index),
    "img_raw":
_bytes_feature(img_raw),
}))
writer.write(example.SerializeToString()) #序列化为字符串
writer.close()
```

此时发现，在指定的目录下多了一个 `dog_train.tfrecords` 文件，如图 8-3 所示。

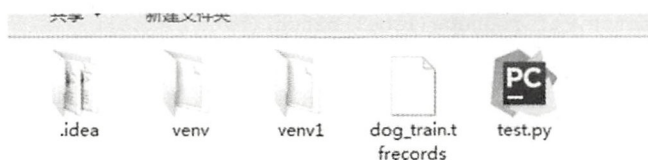


图 8-3

当 TFRecords 格式的文件生成之后，就可以使用队列的形式将其读取出来，再将读取出来的数据运用在实际训练环境当中。一般由 `tf.TFRecordReader()` 读取 TFRecords 格式文件，代码如下：

```
# -*- coding: utf-8 -*-
import os
import tensorflow as tf
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np

#%%
def read_and_decode(filename, batch_size): # read train.tfrecords
    filename_queue = tf.train.string_input_producer([filename]) # create a queue

    reader = tf.TFRecordReader()
    _, serialized_example = reader.read(filename_queue) #return file_name and file
    features = tf.parse_single_example(serialized_example,
        features={
            'label': tf.FixedLenFeature([], tf.int64),
```

```
        'img_raw': tf.FixedLenFeature([], tf.string),
    })#return image and label

    img = tf.decode_raw(features['img_raw'], tf.uint8)
    img = tf.reshape(img, [208, 208, 3]) #reshape image to 512*80*3
    label = tf.cast(features['label'], tf.int32) #throw label tensor

    img_batch, label_batch = tf.train.shuffle_batch([img, label],
                                                    batch_size= batch_size,
                                                    num_threads=64,
                                                    capacity=2000,
                                                    min_after_dequeue=1500,
                                                    )

    return img_batch, tf.reshape(label_batch, [batch_size])

#%%
tfrecords_file = dog_train.tfrecords'
BATCH_SIZE = 20
image_batch, label_batch = read_and_decode(tfrecords_file, BATCH_SIZE)

with tf.Session() as sess:

    i = 0
    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(coord=coord)

    try:
        while not coord.should_stop() and i<1:
            # just plot one batch size
            image, label = sess.run([image_batch, label_batch])
            for j in np.arange(BATCH_SIZE):
                print('label: %d' % label[j])
                plt.imshow(image[j,:,:,:])
                plt.show()
            i+=1
    except tf.errors.OutOfRangeError:
        print('done!')
    finally:
        coord.request_stop()
```


训练结束后就可以输出每张图像所属的分类及图像本身，如图 8-4、图 8-5 所示。

```
label: 0
label: 0
label: 0
label: 1
label: 0
label: 0
label: 0
label: 0
label: 0
label: 0
label: 1
label: 0
label: 1
PyDev console: using IPython 6.1.0
Python 3.6.2 [Anaconda custom (64-bit)] (default, Sep 19 2017, 08:03:39) [MSC v.1900 64 bit (AMD64)] on win32
```

图 8-4

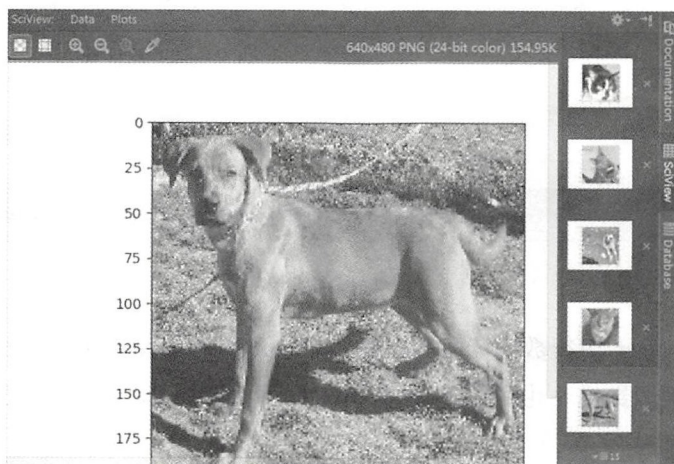


图 8-5

在本段示例代码中，`BATCH_SIZE` 代表要测试输出图片的张数，`plt.show()` 是 Python 中 Matplotlib 库中所带的函数，该函数用于来将图片显示出来，如果没有安装 Matplotlib 库，则需要通过 `pip install matplotlib` 命令安装 Matplotlib 库。

`tf.train.Coordinator()` 和 `tf.train.start_queue_runners()` 函数是 TensorFlow 在处理队列的时候需要用到的相关函数，会在 8.3 节中详细讲解。

8.2 Dataset API 介绍

8.1 节讲解了关于 TFRecords 格式文件的制作和使用方法，这个方法对于处理图像是非常方便的，使用 TFRecords 可以很容易集中管理和操作图像数据。

从 TensorFlow 1.3 开始，谷歌引入 DatasetAPI 作为读取数据的全新方法，从 1.4 版本之后，谷歌又将 Dataset API 从 `tf.contrib.data` 类中提取出来，放入 TensorFlow 的核心类 `tf.data.Dataset` 中。DatasetAPI 从简单的、可重用的片段中构建管道，并进行数据的读取和传输。例如 TensorFlow 在分布式运算环境中，可以使用 Dataset 从集群中的各个节点读取数据进行数据的组装，最后传入使用。

Dataset API 的类图如图 8-6 所示。

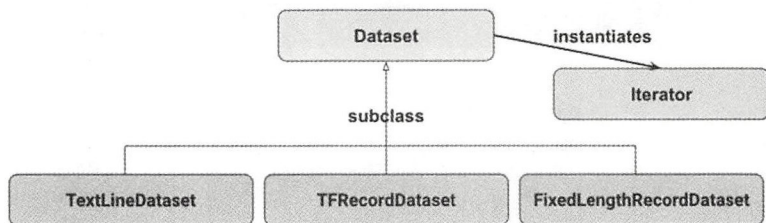


图 8-6

在使用 Dataset 进行数据读取的时候，可以调用 `tf.data.TFRecordDataset` 方法从 TFRecords 中读取数据，也可以调用 `tf.data.TextLineDataset` 方法从文本文件中读取数据。

`tf.data.TFRecordDataset` 方法的定义如下：

```
__init__(
    filenames,
    compression_type=None,
    buffer_size=None
)
```

在 `tf.data.TFRecordDataset` 方法中，需要初始化的有三个参数：`filenames`，`compression_type`，`buffer_size`，其中：

- **filenames**: 包含一个或多个文件名的 `tf.string` 类型的张量。
- **compression_type**: 一个可选参数, 参数值是一个 `tf.string` 类型的标量, 可以向其中传入无压缩、`gzip`、`zlib` 三种压缩格式。
- **buffer_size**: 一个可选参数, 参数值为 `tf.int64` 类型的标量, 此参数用来表示需要缓冲的字节数, 如果值为 0, 则其数值根据压缩类型的默认数值决定。

下面举一个简单的例子说明其具体使用方式。

```
filenames = ["/var/data/files1.tfrecord", "/var/data/files2.tfrecord"]
dataset = tf.data.TFRecordDataset(filenames)
```

首先定义了一个变量, 变量里有两个 `TFRecords` 格式的文件, 再调用 `tf.data.TFRecordDataset` 方法将其传入, 得到一个新的变量 `dataset`。

`tf.data.TextLineDataset` 方法的使用与 `tf.data.TFRecordDataset` 类似, 主要用于生成包含一个或多个文本行的数据集, 定义如下:

```
__init__(
    filenames,
    compression_type=None,
    buffer_size=None
)
```

可以看到, `tf.data.TextLineDataset` 与 `tf.data.TFRecordDataset` 的定义是一模一样的, 在使用过程中唯一不同的是, `tf.data.TFRecordDataset` 传入的文件是 `TFRecords` 格式的文件, 而 `tf.data.TextLineDataset` 传入的是文本格式的文件。由于其使用方法相同, 在此就不重复举例。

除了上述两个比较常用的读取方式外, TensorFlow 还提供了另一种读取方法, 即 `tf.data.FixedLengthRecordDataset` 方法, 这个函数的输入是一个文件的列表和一个 `record_bytes`, 之后 `dataset` 的每一个元素就是文件中固定字节数 `record_bytes` 的内容。通常用来读取以二进制形式保存的文件, 如 `CIFAR10` 数据集就是这种形式。其定义如下:

```
__init__(
    filenames,
    record_bytes,
```

```
header_bytes=None,  
footer_bytes=None,  
buffer_size=None  
)
```

下面对传入参数进行解读：

- **filenames**：包含一个或多个文件名的 `tf.string` 类型的张量；
- **record_bytes**：其类型为一个 `tf.int64` 类型的标量，表示每个记录集中的字节数；
- **header_bytes**：一个可选参数，其类型为一个 `tf.int64` 类型的标量，表示在文件开始时要跳过的字节数；
- **footer_bytes**：一个可选参数，其类型为一个 `tf.int64` 类型的标量，表示在文件末尾要跳过的字节数；
- **buffer_size**：一个可选参数，其类型为一个 `tf.int64` 类型的标量，表示读取时要缓冲的字节数。

在 `tf.data` 下还有一个很重要的类，即 `Iterator`，用来表示迭代数据集的状态。当创建了一个 `Dataset` 数据集之后，一般会再实例化一个 `Iterator` 对象，用来将 `Dataset` 数据集集中的数据取出。在 TensorFlow 中，使用 `dataset.make_initializable_iterator()` 方法来实例化一个 `Iterator` 对象。

下面通过一个简单的官方例子来理解一下 `Iterator` 与 `Dataset` 对象是如何配合使用的。

```
dataset = tf.data.Dataset.range(100)  
iterator = dataset.make_one_shot_iterator()  
next_element = iterator.get_next()  
  
for i in range(100):  
    value = sess.run(next_element)  
    assert i == value
```

在这个例子中，使用 `tf.data.Dataset.range` 方法生成了一个数字的数据集，这里的 `range` 方法与 Python 中的 `range` 方法类似，然后使用 `dataset.make_one_shot_iterator()` 初始化一个 `iterator`，接下来通过 `iterator.get_next()` 从 `iterator` 中取出一个元素，然后通过 `sess.run` 方法运行出来。这个例子中使用的形式被谷歌称为“一次迭代器”，一次

迭代器是最简单的迭代器形式，它只支持通过数据集迭代一次，而不需要显式地初始化。一次迭代器能处理几乎现有的基于队列的输入流水线支持的所有情况，但是它们不支持参数化。

8.3 TensorFlow 中的队列

在使用 TensorFlow 进行异步计算时，队列是一种非常强大的机制。正如 TensorFlow 中的其他组件一样，队列就是 TensorFlow 图中的节点，这是一种有状态的节点，就像变量一样，其他节点可以修改其内容。在 TensorFlow 中，一般可以分成如下四种类型：

(1) **tf.FIFOQueue**：按入列顺序出列的队列，也就是常说的先进先出队列；

(2) **tf.RandomShuffleQueue**：随机顺序出列的队列；

(3) **tf.PaddingFIFOQueue**：以固定长度批量出列的队列，支持通过填充来分配可变大小的张量，PaddingFIFOQueue 可能包含具有动态形状的组件，同时也支持 `dequeue_many`，有兴趣的读者可以查看其构造手册来了解更多内容。

(4) **tf.PriorityQueue**：带优先级出列的队列。

队列本身也是图中的一个节点，所以操作队列的时候可以像操作图中的变量一样去修改队列节点中的内容。下面创建一个最简单的先进先出队列。

```
import tensorflow as tf

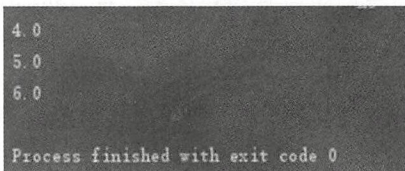
q = tf.FIFOQueue(3, tf.float32)
init = q.enqueue_many([4, 5, 6],)
x = q.dequeue()

with tf.Session() as sess:
    sess.run(init)
    for j in range(sess.run(q.size())):
        print(sess.run(q.dequeue()))
```

简单说一下这里所涉及的一些函数：

- **tf.FIFOQueue**: 这个函数会生成一个先进先出队列，在这里设定其有 3 个值。
- **q.enqueue_many**: 此函数的目的是进行入列操作，在这里入列了 3 个元素，分别是 4.0,5.0,6.0。
- **q.dequeue()**: 此函数的目的是进行出列操作，也就是将入列的元素按照一定的顺序出列，由于使用的是先进先出队列函数，所以如果运行正常就会按照入列的顺序进行出列。

执行完上述程序之后，就打印出队列的结果，而出队列的顺序应该是和入队列的顺序是一样的，如图 8-7 所示。



```
4.0
5.0
6.0

Process finished with exit code 0
```

图 8-7

下面将这个程序做一个简单的修改，看看会发生什么样的变化。

```
import tensorflow as tf

q = tf.FIFOQueue(3, tf.float32)
init = q.enqueue_many([4, 5, 6],)

x = q.dequeue()
y = x+1
q_inc = q.enqueue(y)

with tf.Session() as sess:
    sess.run(init)
    for i in range(3):
        sess.run(q_inc)

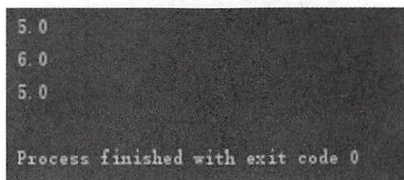
    for j in range(sess.run(q.size())):
        print(sess.run(q.dequeue()))
```

在这段程序中，增加了：

```
y = x+1
```

```
q_inc = q.enqueue(y)
```

这两条语句的主要目的就是出列的元素加 1 后再进行入列操作，当执行这个 session 的时候，将 `q_inc` 执行了 3 次，最后实现的效果如图 8-8 所示。



```
5.0
6.0
5.0

Process finished with exit code 0
```

图 8-8

随机队列和先进先出队列在用法上是比较类似的，下面来看一下随机队列如何使用。

```
import tensorflow as tf

q = tf.RandomShuffleQueue(capacity=10, min_after_dequeue=3, dtypes=tf.float32)

with tf.Session() as sess :
    for i in range(0,10): # 10 次入队
        sess.run(q.enqueue(i))

    for i in range(0,7): # 7 次出队
        print(sess.run(q.dequeue()))
```

此段代码展示了如何使用 TensorFlow 建立一个随机队列。首先使用 `tf.RandomShuffleQueue(capacity=10, min_after_dequeue=3, dtypes=tf.float32)` 方法建立了一个随机队列，并定义了最大长度为 10，出列后最小长度为 3，数据类型为 `tf.float32`。

在运行 session 阶段，我们为其指定了 10 次入队列操作，再进行 7 次出队列操作，由于是随机队列，所以输出的结果也是随机数列，最后打印出来的结果如图 8-9 所示。

接下来思考一个问题，如果将出列的次数由 7 次改成 8 次，会怎么样呢？事实上，如果将出列的长度改成 8 后，打印出来的结果仍然是 7 次，这是因为在定义这个随机队列的时候指定其出列后最小长度为 7。

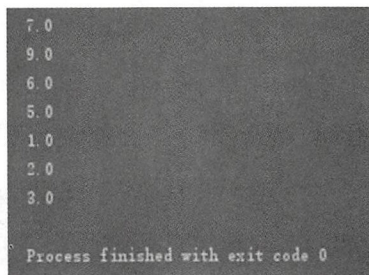


图 8-9

一般来讲，Python 常用的线程技术都是单线程操作，那么 TensorFlow 对于多线程是如何进行管理和操作的呢？实际上，TensorFlow 为我们提供了 `tf.Coordinator` 和 `tf.QueueRunner` 两个类来实现多线程的管理功能，`tf.Coordinator` 类主要是用来同时停止多个工作线程并且向那个在等待所有工作线程终止的程序报告异常的，`tf.QueueRunner` 类主要用来协调多个工作线程同时将多个张量推入同一个队列中。

在使用 `tf.Coordinator` 类的时候，主要会用到以下三个方法：

- **should_stop**: 如果线程应该停止，则返回 `True`。
- **request_stop(<exception>)**: 请求该线程停止。
- **join(<list of threads>)**: 等待被指定的线程终止。

`tf.QueueRunner` 一般需要与 `Queue` 一起使用，但是在使用 `tf.QueueRunner` 的时候并不必须要使用 `tf.Coordinator` 类。例如：

```
import tensorflow as tf

q = tf.FIFOQueue(10, "float")
counter = tf.Variable(0.0) #计数器
# 给计数器加1
increment_op = tf.assign_add(counter, 1.0)
# 将计数器加入队列
enqueue_op = q.enqueue(counter)

qr = tf.train.QueueRunner(q, enqueue_ops=[increment_op, enqueue_op] * 2)

# 主线程
sess = tf.InteractiveSession()
```



```
tf.global_variables_initializer().run()  
# 启动入队线程  
qr.create_threads(sess, start=True)  
for i in range(20):  
    print (sess.run(q.dequeue()))
```

在这段代码中，我们首先定义了一个先进先出的队列，然后创建了一个计数器，并使计数器执行了一个加 1 的操作，然后将计数器入列。之后使用 `tf.train.QueueRunner` 函数创建了一个 `QueueRunner` 命令，并使用多线程的方式向其添加了数据，之后又创建了 4 个线程，这里包含了两个计数线程和两个执行入队列线程，最后调用 `run` 方法将线程启动，执行效果如图 8-10 所示。

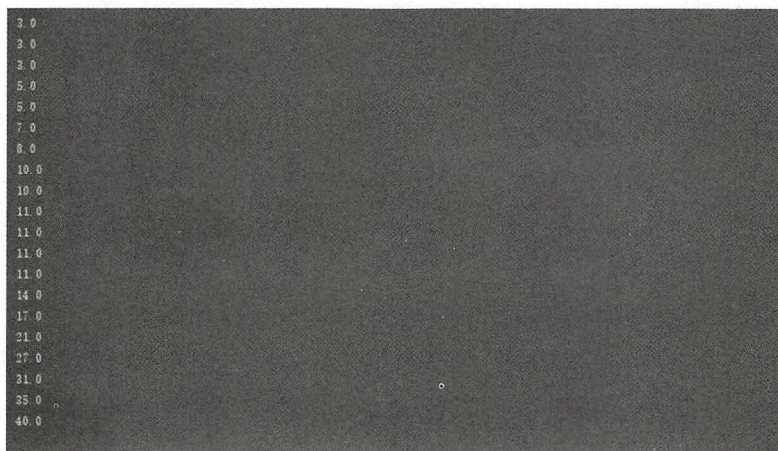


图 8-10

仔细观察输出结果可以发现，其实到目前为止，程序并没有结束执行，而是在某个地方停止了。在上述的操作中，执行入队的进程会先执行 10 次，执行 10 次是由队列长度的限制所导致的。在执行完 10 次操作之后，主线程就会开始消耗数据，当一部分数据被消耗后，入队的进程又会继续开始执行。最后当主线程消耗完 20 个数据后就会停止，但这个停止并不会去影响其他线程的执行，因此程序并不会结束。

接下来使用 `tf.Coordinator` 类来控制多个线程的终止和开始操作，以避免上述问题的发生。在使用 `tf.Coordinator` 类的时候，首先要创建一个 `Coordinator` 对象，然后建立一些能够使用 `Coordinator` 对象的线程，通常这些线程会一直循环运行，直到

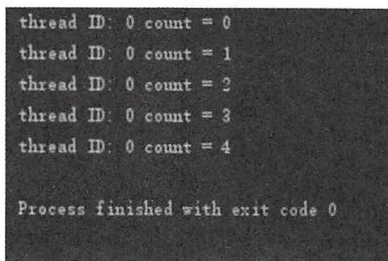
`should_stop()`函数返回 `True` 才会停止。任何一个线程只需要调用 `request_stop()`函数，同时其他线程的 `should_stop()`函数将返回 `True`，就能决定线程的停止时间。再看另外一段示例代码：

```
#encoding=utf-8
import threading
import tensorflow as tf

def func(coord, t_id):
    count = 0
    while not coord.should_stop():
        print('thread ID:',t_id, 'count =', count)
        count += 1
        if(count == 5):
            coord.request_stop()
coord = tf.train.Coordinator()
threads = [threading.Thread(target=func, args=(coord, i)) for i in range(4)]

for t in threads:
    t.start()
coord.join(threads)
```

首先创建一个函数实现多线程，参数为 `Coordinator` 和线程号，使用 `while not coord.should_stop()`来执行不应该停止计数的处理过程，当计数器计数到 5 的时候，就调用 `Coordinator` 类的 `request_stop()`函数请求终止线程，最后使用 `coord.join(threads)`来表示等待所有线程的结束，运行结果如图 8-11 所示。



```
thread ID: 0 count = 0
thread ID: 0 count = 1
thread ID: 0 count = 2
thread ID: 0 count = 3
thread ID: 0 count = 4

Process finished with exit code 0
```

图 8-11

当线程计数器执行完前 4 步的时候，所有的线程将会停止，整个程序运行结束，这是因为我们在程序中使用了 `Coordinator` 类的 `request_stop()`函数。

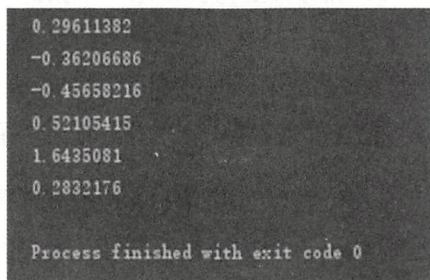
已知 QueueRunner 类和 Coordinator 类的用途及使用方法后，可以尝试着将这两个类结合使用，用 QueueRunner 类来创建队列，用 Coordinator 类来控制队列中线程的停止和开始，由此来看看它们是如何进行配合的，代码如下：

```
import tensorflow as tf

queue = tf.FIFOQueue(100, 'float')
enqueue_op = queue.enqueue(tf.random_normal([1]))
qr = tf.train.QueueRunner(queue, [enqueue_op] * 5)
tf.train.add_queue_runner(qr)
out_tensor = queue.dequeue()

with tf.Session() as sess:
    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(sess=sess, coord=coord)
    for i in range(6):
        print(sess.run(out_tensor) [0])
    coord.request_stop()
    coord.join(threads)
```

首先使用 `tf.FIFOQueue(100, 'float')` 创建了一个先进先出队列，然后使用 `queue.enqueue(tf.random_normal([1]))` 函数将其入列，入列的值为随机数，接下来执行 `tf.train.QueueRunner()` 函数启动 5 个线程，并使用 `tf.train.add_queue_runner()` 将线程添加到图中。在运行图的时候，程序从第一个计数器开始，每个计数器都会将出列的内容打印出来，直到计数器执行到第 6 次时停止所有的线程。至此程序运行结束，运行结果如图 8-12 所示。



```
0.29611382
-0.36206686
-0.45658216
0.52105415
1.6435081
0.2832176

Process finished with exit code 0
```

图 8-12

第 9 章

支持向量机 (SVM)

在使用深度学习进行垃圾邮件分类、图像识别及解决其他分类问题时都离不开统计学，在统计学中有很多分类方法，其中比较常用的、也比较强大的一种分类方法或分类器，就是支持向量机。

9.1 什么是支持向量机

支持向量机 (Support Vector Machine, SVM) 是一种监督学习方法，在统计分类和回归分析中应用得非常广泛。给定一组训练实例，每个训练实例被标记为属于两个分类中的一个或另一个，SVM 训练算法将会创建一个新的实例分配给两个类别之一的模型，使其成为非概率二元线性分类器。SVM 模型是将实例表示为空间中的点，这样映射就使得单独类别的实例被宽得尽可能明显的间隔分开。然后，将新的实例映射到同一空间，并基于它们落在间隔的哪一侧来预测所属类别。

除进行线性分类之外，SVM 还可以使用所谓的核技巧有效地进行非线性分类、输入隐式映射到高维特征空间中。当数据未被标记，不能进行监督学习，而需要用非监督学习时，它会尝试找出数据到簇的自然聚类，并将新数据映射到这些已形成的簇。这种将支持向量机改进后的聚类算法称为支持向量聚类。当数据未被标记或者部分数据被标记时，支持向量聚类经常在工业应用中用于分类的预处理。

在众多知名的数据挖掘算法中,支持向量机 (SVM) 有着非常重要的地位,可以说,支持向量机是最健壮、最准确的数据挖掘算法之一。SVM 属于二分类算法,可以支持线性和非线性的分类。如图 9-1 所示为一个使用 SVM 算法处理的二分类散点数据。

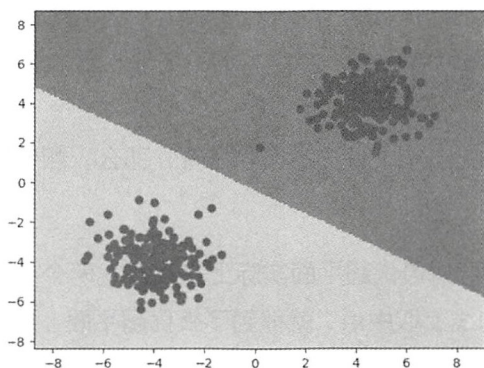


图 9-1

由图 9-1 可以发现,散落着的右上方圆点和左下方圆点被一条很明显的界限分隔开,且从这条分隔线开始,向左下角点和右上角点的距离也是最大的,这也就是 SVM 的目的所在,即找到一个超平面使样本分成两类,并且间隔最大。如果将这个问题看作一个线性问题 $y=Wx+b$, 所求得的 W 就代表着我们需要寻找的超平面系数。

那么,如何才能找到最优的超平面系数呢? 如图 9-2 所示。

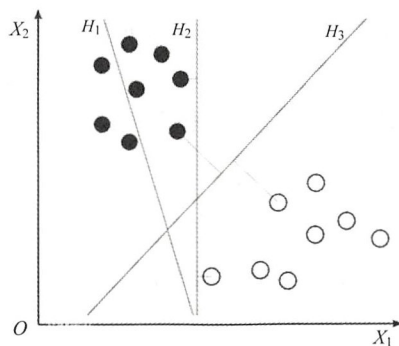


图 9-2

在图 9-2 中, H_2 和 H_3 都可以作为超平面, 因为这两条线都能够很清楚地将黑色的圆圈和白色的圆圈区分出来。所以说, 在解决分类问题中, 超平面可能不止一个, 但是最优超平面只有一个, 正如图 9-2 所示。我们很清楚地就能知道 H_3 是图 9-2 中最优的超平面, 因为只有这一条线才使得黑色圆圈和白色圆圈之间的间隔最大。

9.2 计算最优超平面

计算出最优超平面是 SVM 算法的核心任务, 那么, 如何才能计算并找到最优的超平面呢?

根据图 9-2, 我们寻找最优超平面实际上是在寻找两个点之间的最大距离, 在两个点之间的最大距离的线上取中点, 就得到了最优超平面。我们可以把这个分类超平面设为 $f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b = 0$, 将样本设置为 0, 把 \mathbf{x} 代入到函数 $f(\mathbf{x})$ 中, 如果得出的 $f(\mathbf{x})$ 的结果小于 0, 就对该样本标记一个 -1 的类别标签 y_i ; 如果结果大于 0, 就对该样本标记一个 +1 的 y_i 。我们可以将 ± 1 理解为两个分类, 之所以这样标记, 是因为可以方便地求导, 由此就可以定义一个函数的间隔, 如下:

$$\hat{\gamma} = |\mathbf{w}^T \mathbf{x} + b| = y_i (\mathbf{w}^T \mathbf{x} + b)$$

因为当 $f(\mathbf{x})$ 的结果小于 0 时, 所得到的样本会是一个负数, 所以在前面乘上一个 y_i 可以保证这个间隔的非负性。

一般来讲, 求 $f(\mathbf{x})$ 的过程实际上就是求参数 \mathbf{w} 和 b 的过程, 其中 \mathbf{w} 代表着一个 n 维向量, b 代表一个实数, 当我们把 \mathbf{w} 求出之后, 就可以根据样本的具体数值将该点代入求得 b 。因此, 在进行求解的时候需要更多地关注这个 \mathbf{w} 该如何求得, 因为 \mathbf{w} 才是变量。

如图 9-3 所示, 对任意不在分类超平面上的点 x_i , 我们可以依赖它到分类超平面的垂直投影 x_0 , 从而计算出它到分类超平面上的几何间隔:

$$\hat{\gamma} = \frac{|\mathbf{w}^T \mathbf{x} + b|}{\|\mathbf{w}\|} = \frac{y_i (\mathbf{w}^T \mathbf{x} + b)}{\|\mathbf{w}\|} = \frac{\hat{\gamma}}{\|\mathbf{w}\|}$$

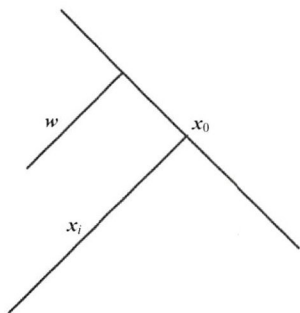


图 9-3

在这里, $\|w\|$ 代表向量 w 的范数, 是对 w 长度的一种度量。我们将间隔表示为 M , 一般来讲, 间隔的取值是最近距离的 2 倍, 于是有:

$$M = \frac{2}{\|w\|}$$

最大化这个式子等价于最小化 $\|w\|$, 另外由于 $\|w\|$ 是一个单调函数, 我们可以加入平方和前面的系数, 以利于求导。于是就有:

$$\max \frac{1}{\|w\|} \rightarrow \min \frac{\|w\|^2}{2}$$

这样就将这个问题转换成一个等价的二次规划问题, 必然能够得到一个全局的最优解。

9.3 TensorFlow 实现线性 SVM

通过前面两节的讲解, 我们了解了线性 SVM 的基本特性, 以及用公式求得全局最优超平面的方法, 下面, 要用 TensorFlow 代码实现一个线性的支持向量机。

首先来分析一下我们需要做哪些准备:

(1) 定义一批数据用于训练和验证;

- (2) 使用线性回归的方式训练这批数据;
- (3) 训练时求出每一步的损失值;
- (4) 使用训练得到的方法进行验证。

具体代码如下:

```
import tensorflow as tf
import numpy as np
from matplotlib import pyplot as plt

class SVM():
    def __init__(self):
        self.x=tf.placeholder('float',shape=[None,2],name='x_batch')
        self.y=tf.placeholder('float',shape=[None,1],name='y_batch')
        self.sess=tf.Session()

    def creat_dataset(self,size, n_dim=2, center=0, dis=2, scale=1,
one_hot=False):
        center1 = (np.random.random(n_dim) + center - 0.5) * scale + dis
        center2 = (np.random.random(n_dim) + center - 0.5) * scale - dis
        cluster1 = (np.random.randn(size, n_dim) + center1) * scale
        cluster2 = (np.random.randn(size, n_dim) + center2) * scale
        x_data = np.vstack((cluster1, cluster2)).astype(np.float32)
        y_data = np.array([1] * size + [-1] * size)
        indices = np.random.permutation(size * 2)
        x_data, y_data = x_data[indices], y_data[indices]
        y_data=np.reshape(y_data,(y_data.shape[0],1))
        if not one_hot:
            return x_data, y_data
        y_data = np.array([[0, 1] if label == 1 else [1, 0] for label in y_data],
dtype=np.int8)
        return x_data, y_data

    @staticmethod
    def get_base(self,_nx, _ny):
        _xf = np.linspace(self.x_min, self.x_max, _nx)
        _yf = np.linspace(self.y_min, self.y_max, _ny)
        n_xf, n_yf = np.meshgrid(_xf, _yf)
        return _xf, _yf,np.c_[n_xf.ravel(), n_yf.ravel()]
```



```

def train(self, step, x_data, y_data):

    w = tf.Variable(np.ones([2,1]), dtype=tf.float32, name="w_v")
    b = tf.Variable(0., dtype=tf.float32, name="b_v")

    self.y_pred = tf.matmul(self.x, w) + b

    cost = tf.nn.l2_loss(w) + tf.reduce_sum(tf.maximum(1 - self.y * self.y_pred, 0))
    train_step = tf.train.AdamOptimizer(0.01).minimize(cost)

    self.y_predict = tf.sign(tf.matmul(self.x, w) + b)
    self.sess.run(tf.global_variables_initializer())
    for i in range(step):
        index = np.random.permutation(y_data.shape[0])
        x_data1, y_data1 = x_data[index], y_data[index]
        self.sess.run(train_step, feed_dict={self.x: x_data1[0:50], self.y: y_data1[0:50]})
        self.y_predict_value, self.w_value, self.b_value, cost_value = self.sess.run([self.y_predict, w, b, cost], feed_dict={self.x: x_data, self.y: y_data})
        if i % 1000 == 0: print('cost=%f' % cost_value)
    def predict(self, y_data):

        correct = tf.equal(self.y_predict_value, y_data)

        precision = tf.reduce_mean(tf.cast(correct, tf.float32))

        precision_value = self.sess.run(precision)
        return precision_value, self.y_predict_value

    def drawresult(self, x_data):

        self.x_min, self.y_min = np.minimum.reduce(x_data, axis=0) - 2
        self.x_max, self.y_max = np.maximum.reduce(x_data, axis=0) + 2

        xf, yf, matrix_ = self.get_base(self, 200, 200)

        print(self.w_value, self.b_value)
        z = np.sign(np.matmul(matrix_, self.w_value) + self.b_value).reshape((200, 200))
        plt.pcolormesh(xf, yf, z, cmap=plt.cm.Paired)

```



```
for i in range(x_data.shape[0]):

    if self.y_predict_value[i,0]==1.0:
        plt.scatter(x_data[i,0],x_data[i,1],color='r')
    else:
        plt.scatter(x_data[i,0],x_data[i,1],color='g')

plt.axis([self.x_min,self.x_max,self.y_min ,self.y_max])
# plt.contour(xf, yf, z)
plt.show()

svm=SVM()
x_data,y_data=svm.creat_dataset(size=200, n_dim=2, center=0, dis=4,
one_hot=False)

svm.train(5000,x_data,y_data)
precision_value,y_predict_value=svm.predict(y_data)
svm.drawresult(x_data)
```

上述代码运行出来的结果如图 9-4 所示。

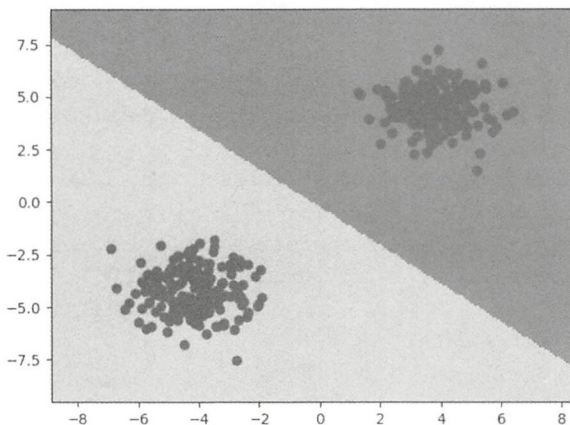


图 9-4

下面来解释一下在上述代码中用到的几个新知识及常用的函数。首先，我们引入了两个库：NumPy 和 Matplotlib。其中 NumPy 是一个用 Python 实现的科学计算包，包括一个强大的 N 维数组对象 Array、比较成熟的（广播）函数库、用于整合 C/C++ 和 Fortran 代码的工具包，以及实用的线性代数、傅立叶变换和随机数生成函数。

Matplotlib 是一个 Python 的 2D 绘图库，它以各种硬拷贝格式和跨平台的交互式环境生成出版质量级别的图形。

我们使用了 NumPy 库中的 `np.random.random()` 函数来随机创建一些数据。

在训练时，我们定义了 5 000 个训练步数，并在每 1 000 步的时候打印出一个损失值。

训练结束后，我们使用另一批数据进行预测，并使用 Matplotlib 库将预测的结果打印出来，最后形成如图 9-4 所示的结果。

9.4 非线性 SVM 介绍

前面讲解了线性的 SVM 分类器，接下来讲解非线性 SVM 分类器的一些基本特征。

在我们的生活和实际应用过程中，不可能所有的问题都是线性问题，确切地说，我们所面对的非线性问题要远远多于线性问题。

图 9-5 所示就是一个典型的非线性分类问题，我们无法使用一条直线将图中的实心点和空心点分开，甚至说即使使用一条曲线也很难将其进行一个非常明显的划分，针对这类问题，需要用非线性 SVM 来解决。

先将问题简化一下，图 9-6 也是一个很典型的线性不可分的图像，也就是说，它也是一个非线性的数据，但是如果在这个 xy 的二维平面图中加上一个 z 轴，形成一个三维的空间图像，那么这个问题又会变成什么呢？

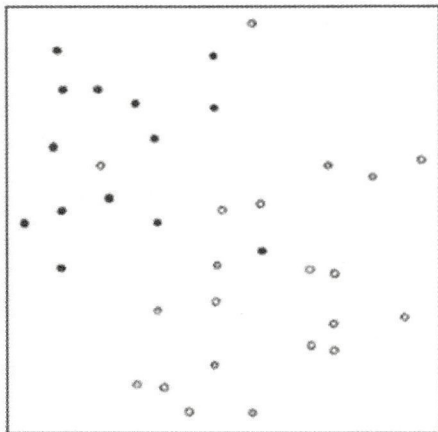


图 9-5

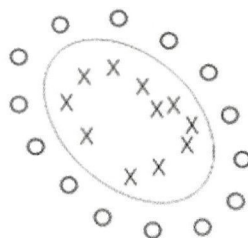


图 9-6

图 9-7 在图 9-6 的基础上增加了一个 z 轴，而这个 z 轴的目的就是将 \circ 和 \times 使用一个分离超平面进行分离，如果使用 SVM 算法在空间上将其分离，那么这个问题是不是就迎刃而解了呢？

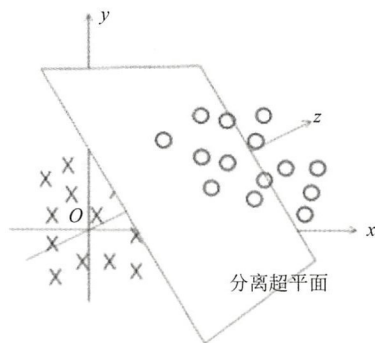


图 9-7

首先这是一个非常不错的想法，但是实现起来其实并没有那么简单，为了解决这个问题，我们在此引入了一个新的方法——核函数。从本质上讲，我们可以将核函数定义为一个从低维度到高维度的映射，这个核所定义的映射可以理解为点对点的映射。

核函数的优点是可以在空间中直接计算内积 $\langle \phi(x)_i, \phi(x) \rangle$ ，就像原始输入点的函数



一样,有可能将两个步骤融合到一起,从而建立一个非线性的学习器。用 K 来描述一下核函数,对于所有的 x 轴和 z 轴上的(这里假设 x 和 z 都是 n 维的)数据 X , X 属于 $R(n)$ 空间,非线性函数 ϕ 实现输入空间 x 到特征空间 F 的映射,其中 F 属于 $R(m), n < m$, 则 $K(x, z) = \langle \phi(x) \cdot \phi(z) \rangle$, 这里的 ϕ 是从 x 到内积特征空间 F 的映射。从这个函数中可以看出,核函数将 m 维高维空间的内积运算转化 n 维低维输入空间的核函数计算,巧妙地解决了在高维特征空间中的“维数灾难”等问题,从而为高维特征空间解决复杂的分类问题或回归问题奠定了理论基础。

在实际的应用过程中,我们可以选择的核函数有很多种,例如高斯核、多项式核等,下面就来简单地介绍一下这些核函数的概念及应用场景。

高斯核

高斯核又称径向基函数核,或称 RBF 核,是机器学习中一种常用的核函数,是支持向量机分类中最常用的核函数之一。其函数表达式为: $K(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right)$, 在这里,设定 $\gamma > 0$, $\|x - x'\|^2$ 可以看作两个特征向量的平方欧几里得距离, σ 是一个自由参数。一般来讲,我们可以设置一种等价但更为简单的参数 γ , 将其表达式设为 $\gamma = \frac{1}{2\sigma^2}$, 于是就有 $K(x, x') = \exp(-\gamma \|x - x'\|^2)$, 这也是高斯核的一种最常见的表达式。

高斯核会将原始空间映射为无穷维空间,不过,如果 σ 选得非常大的话,高斯特征上的权重实际上就会衰减得非常快,因此,可以将其看作一个低维的自空间;但是如果 σ 选得非常小的话,则可以将任意的数据映射为线性可分,但这并不一定是好事,因为如果 σ 选得非常小的话,也很有可能产生非常严重的过拟合问题。

如图 9-8 所示, $-1, -10, -100$ 是 γ 的三个选值,当然,在这里也可以看作为 $\gamma = \frac{1}{2\sigma^2}$ 选了三个值,当 γ 变大的时候,就相当于 σ 会变小,此时的线性组合曲线就会变得不均匀,且不平滑,甚至会产生孤岛,因此,在选择参数 γ 的时候,通常不建议使用太大的 γ 。



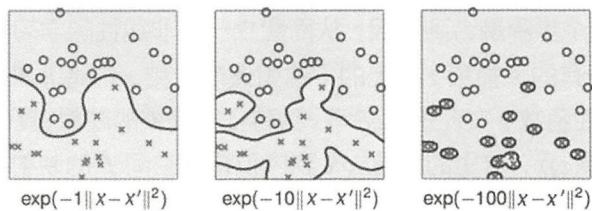


图 9-8

多项式核

多项式核也是 SVM 中常见的核函数之一，有着良好的全局性质，阶数越低，其推广能力越强。多项式核对于线性不可分问题也有着很强的解决能力。多项式核的表达式为：

$$K(x, x_i) = ((x \cdot x_i) + 1)^d$$

多项式核能够很容易地将低维的输入空间映射到高维的特征空间，但是如果多项式核的参数非常多且阶数非常高的话，整个函数的值将会趋向于无限大或无限小，计算的复杂程度将会变得无穷大，甚至无法计算。

9.5 使用 TensorFlow 实现非线性 SVM 分类器

9.4 节讲解了非线性 SVM 分类器的相关特性，并介绍了非线性 SVM 分类器的相关原理，也了解了核函数的相关特性，以及不同核函数的区别。在本节中，我们将使用 TensorFlow 完成一个基于高斯核的非线性 SVM 分类器。

在看具体代码之前先来介绍一个新的模块——Sklearn。Sklearn 是基于 NumPy 和 Scipy 的一个机器学习算法库，也是机器学习领域中的一个常用的 Python 第三方模块。使用 Sklearn 可以很方便地调用常用的机器学习算法，并且只需要进行简单的调用就可以完成大多数机器学习任务。Sklearn 主要包括分类、回归、降维和聚类四大机器学习算法，还包含特征提取、数据处理和模型评估三大模块。

另外，这里使用的 IRIS 数据集是常用的分类实验数据集，由 Fisher 于 1936 年收



集整理。IRIS 也称鸢尾花卉数据集，是一类多重变量分析的数据集，包含 150 个数据子集，分为 3 类，每类 50 个数据，每个数据包含 4 个属性，其中的一类与另外两类是线性可分离的，后两类是非线性可分离的。可通过花萼长度、花萼宽度、花瓣长度、花瓣宽度 4 个属性预测鸢尾花卉属于 Setosa, Versicolour, Virginica 三类中的哪一类。

我们所用到的高斯核公式为

$$K(x_1, x_2) = \exp(-\gamma |x_1 - x_2|^2)$$

具体实现代码如下：

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets
from tensorflow.python.framework import ops
ops.reset_default_graph()

# Create graph
sess = tf.Session()

# Load the data
# iris.data = [(Sepal Length, Sepal Width, Petal Length, Petal Width)]
# 加载 IRIS 数据集，抽取花萼长度和花瓣宽度，分隔每类的 x_vals 值和 y_vals 值
iris = datasets.load_iris()
x_vals = np.array([[x[0], x[3]] for x in iris.data])
y_vals = np.array([1 if y==0 else -1 for y in iris.target])
class1_x = [x[0] for i,x in enumerate(x_vals) if y_vals[i]==1]
class1_y = [x[1] for i,x in enumerate(x_vals) if y_vals[i]==1]
class2_x = [x[0] for i,x in enumerate(x_vals) if y_vals[i]==-1]
class2_y = [x[1] for i,x in enumerate(x_vals) if y_vals[i]==-1]

# Declare batch size
# 声明批量大小（偏向于更大批量）
batch_size = 150

# Initialize placeholders
x_data = tf.placeholder(shape=[None, 2], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
prediction_grid = tf.placeholder(shape=[None, 2], dtype=tf.float32)
```



TensorFlow 进阶指南

基础、算法与应用

```

# Create variables for svm
b = tf.Variable(tf.random_normal(shape=[1, batch_size]))

# Gaussian (RBF) kernel
# 声明批量大小 (偏向于更大批量)
gamma = tf.constant(-25.0)
sq_dists = tf.multiply(2., tf.matmul(x_data, tf.transpose(x_data)))
my_kernel = tf.exp(tf.multiply(gamma, tf.abs(sq_dists)))

# Compute SVM Model
first_term = tf.reduce_sum(b)
b_vec_cross = tf.matmul(tf.transpose(b), b)
y_target_cross = tf.matmul(y_target, tf.transpose(y_target))
second_term = tf.reduce_sum(tf.multiply(my_kernel, tf.multiply(b_vec_cross,
y_target_cross)))
loss = tf.negative(tf.subtract(first_term, second_term))

# Gaussian (RBF) prediction kernel
# 创建一个预测核函数
rA = tf.reshape(tf.reduce_sum(tf.square(x_data), 1), [-1, 1])
rB = tf.reshape(tf.reduce_sum(tf.square(prediction_grid), 1), [-1, 1])
pred_sq_dist = tf.add(tf.subtract(rA, tf.multiply(2., tf.matmul(x_data,
tf.transpose(prediction_grid)))), tf.transpose(rB))
pred_kernel = tf.exp(tf.multiply(gamma, tf.abs(pred_sq_dist)))

# 声明一个准确率函数, 为正确分类的数据点的百分比
prediction_output = tf.matmul(tf.multiply(tf.transpose(y_target), b),
pred_kernel)
prediction = tf.sign(prediction_output - tf.reduce_mean(prediction_output))
accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.squeeze(prediction),
tf.squeeze(y_target)), tf.float32))

# Declare optimizer
my_opt = tf.train.GradientDescentOptimizer(0.01)
train_step = my_opt.minimize(loss)

# Initialize variables
init = tf.global_variables_initializer()
sess.run(init)

# Training loop
loss_vec = []

```




```
batch_accuracy = []
for i in range(300):
    rand_index = np.random.choice(len(x_vals), size=batch_size)
    rand_x = x_vals[rand_index]
    rand_y = np.transpose([y_vals[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})

    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})
    loss_vec.append(temp_loss)

    acc_temp = sess.run(accuracy, feed_dict={x_data: rand_x,
                                              y_target: rand_y,
                                              prediction_grid: rand_x})
    batch_accuracy.append(acc_temp)

    if (i+1)%75==0:
        print('Step #' + str(i+1))
        print('Loss = ' + str(temp_loss))

# Create a mesh to plot points in
# 为了绘制决策边界 (Decision Boundary), 我们创建一个数据点(x,y)的网格, 评估预测函数。
x_min, x_max = x_vals[:, 0].min() - 1, x_vals[:, 0].max() + 1
y_min, y_max = x_vals[:, 1].min() - 1, x_vals[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                     np.arange(y_min, y_max, 0.02))
grid_points = np.c_[xx.ravel(), yy.ravel()]
[grid_predictions] = sess.run(prediction, feed_dict={x_data: rand_x,
                                                    y_target: rand_y,
                                                    prediction_grid: grid_points})
grid_predictions = grid_predictions.reshape(xx.shape)

# Plot points and grid
plt.contourf(xx, yy, grid_predictions, cmap=plt.cm.Paired, alpha=0.8)
plt.plot(class1_x, class1_y, 'ro', label='I. setosa')
plt.plot(class2_x, class2_y, 'kx', label='Non setosa')
plt.title('Gaussian SVM Results on Iris Data')
plt.xlabel('Pedal Length')
plt.ylabel('Sepal Width')
plt.legend(loc='lower right')
plt.ylim([-0.5, 3.0])
plt.xlim([3.5, 8.5])
plt.show()
```




```
# Plot batch accuracy
plt.plot(batch_accuracy, 'k-', label='Accuracy')
plt.title('Batch Accuracy')
plt.xlabel('Generation')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()

# Plot loss over time
plt.plot(loss_vec, 'k-')
plt.title('Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.show()
```

首先使用 `datasets.load_iris()` 方法加载数据, 并将每一类的 `x_vals` 值和 `y_vals` 值分隔开, 然后声明 `batch` 的大小为 150。

```
iris = datasets.load_iris()
x_vals = np.array([[x[0], x[3]] for x in iris.data])
y_vals = np.array([1 if y==0 else -1 for y in iris.target])
class1_x = [x[0] for i,x in enumerate(x_vals) if y_vals[i]==1]
class1_y = [x[1] for i,x in enumerate(x_vals) if y_vals[i]==1]
class2_x = [x[0] for i,x in enumerate(x_vals) if y_vals[i]==-1]
class2_y = [x[1] for i,x in enumerate(x_vals) if y_vals[i]==-1]

batch_size = 150
```

接下来, 初始化输入参数, 并用 `x_data` 和 `y_target` 来表示, 且创建一个随机偏置量。

```
x_data = tf.placeholder(shape=[None, 2], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
prediction_grid = tf.placeholder(shape=[None, 2], dtype=tf.float32)
b = tf.Variable(tf.random_normal(shape=[1, batch_size]))
```

然后声明批量大小, 并创建可用于计算的 SVM 模型, 在这里我们一般会去定义一些用于计算的常量和交叉熵, 使用最常用的 `tf.matmul()` 方法计算交叉熵, 并使用 `tf.reduce_sum` 方法进行压缩求和, 最终生成一个 SVM 模型。



```
gamma = tf.constant(-25.0)
sq_dists = tf.multiply(2., tf.matmul(x_data, tf.transpose(x_data)))
my_kernel = tf.exp(tf.multiply(gamma, tf.abs(sq_dists)))

first_term = tf.reduce_sum(b)
b_vec_cross = tf.matmul(tf.transpose(b), b)
y_target_cross = tf.matmul(y_target, tf.transpose(y_target))
second_term = tf.reduce_sum(tf.multiply(my_kernel, tf.multiply(b_vec_cross,
y_target_cross)))
loss = tf.negative(tf.subtract(first_term, second_term))
```

在定义完模型后，一般还需要定义一个预测核函数，在本例中，我们使用的就是最常用的高斯核函数进行计算。在创建高斯核函数前，我们需要声明一个准确率函数，其目的是提高正确分类的数据点的百分比。

```
prediction_output = tf.matmul(tf.multiply(tf.transpose(y_target), b),
pred_kernel)
prediction = tf.sign(prediction_output - tf.reduce_mean(prediction_output))
accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.squeeze(prediction),
tf.squeeze(y_target)), tf.float32))
最后，定义一个笛卡尔优化器，并进行迭代计算和非线性 SVM 分类运算。
init = tf.global_variables_initializer()
sess.run(init)

loss_vec = []
batch_accuracy = []
for i in range(300):
    rand_index = np.random.choice(len(x_vals), size=batch_size)
    rand_x = x_vals[rand_index]
    rand_y = np.transpose([y_vals[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})

    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})
    loss_vec.append(temp_loss)

    acc_temp = sess.run(accuracy, feed_dict={x_data: rand_x,
                                              y_target: rand_y,
                                              prediction_grid: rand_x})
    batch_accuracy.append(acc_temp)

    if (i+1)%75==0:
        print('Step #' + str(i+1))
```




```
print('Loss = ' + str(temp_loss))
```

此时训练部分就结束了，一般会使用测试数据点验证训练是否准确，并将其绘制出来。

```
x_min, x_max = x_vals[:, 0].min() - 1, x_vals[:, 0].max() + 1
y_min, y_max = x_vals[:, 1].min() - 1, x_vals[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                     np.arange(y_min, y_max, 0.02))
grid_points = np.c_[xx.ravel(), yy.ravel()]
[grid_predictions] = sess.run(prediction, feed_dict={x_data: rand_x,
                                                    y_target: rand_y,
                                                    prediction_grid: grid_points})
grid_predictions = grid_predictions.reshape(xx.shape)

plt.contourf(xx, yy, grid_predictions, cmap=plt.cm.Paired, alpha=0.8)
plt.plot(class1_x, class1_y, 'ro', label='I. setosa')
plt.plot(class2_x, class2_y, 'kx', label='Non setosa')
plt.title('Gaussian SVM Results on Iris Data')
plt.xlabel('Pedal Length')
plt.ylabel('Sepal Width')
plt.legend(loc='lower right')
plt.ylim([-0.5, 3.0])
plt.xlim([3.5, 8.5])
plt.show()

plt.plot(batch_accuracy, 'k-', label='Accuracy')
plt.title('Batch Accuracy')
plt.xlabel('Generation')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()

# Plot loss over time
plt.plot(loss_vec, 'k-')
plt.title('Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.show()
```

最终，我们运行程序后会得到一个结果，如图 9-9 所示。



```
Step #75  
Loss = -120.115  
Step #150  
Loss = -232.615  
Step #225  
Loss = -345.115  
Step #300  
Loss = -457.615
```

图 9-9

这里每隔 75 步打印一次汇总, 最后所绘制的图形如图 9-10 所示。

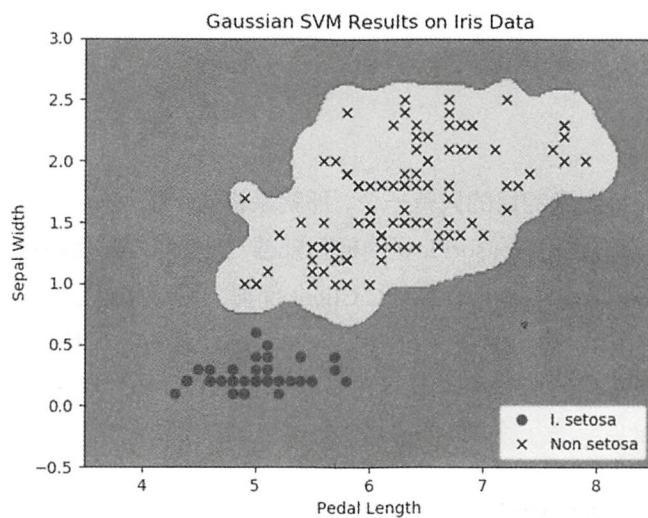


图 9-10

从图 9-10 中可以看出, ●和 × 被明显地分隔开, 很明显不能使用标准的线性 SVM 算法, 应该使用非线性 SVM 算法分隔。

第 10 章

TensorFlow 结合 Flask

发布 MNIST 模型

一个模型被训练出来的目的只有一个，那就是将这个模型运用到实际的生产当中。一般来讲，有两种方式进行 TensorFlow 的模型部署：第一种方式就是谷歌官方推荐的，使用 TensorFlow Serving + Golang GRPC Client 部署；另一种就是使用 Web 应用框架调用 TensorFlow 模型，再封装成 API 接口，供前端界面进行调用。本章使用第二种方式进行部署，所选用的框架是使用 Python 语言编写的 Flask 框架。

10.1 Flask 框架介绍

Flask 是一个使用 Python 语言编写的轻量级 Web 应用框架。其 WSGI 工具箱采用 Werkzeug，模板引擎则使用 Jinja2。Flask 使用 BSD 授权。Flask 也被称为“Microframework”，因为它使用简单的核心，用 extension 增加其他功能。Flask 没有默认使用的数据库、窗体验证工具。

下面展示一个最简单的 Flask 应用，以提高大家对 Flask 框架的了解。

```
from flask import Flask
```

```
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run()
```

将程序放入 IDE 环境中或直接保存成以 py 为后缀名的文件，然后运行，出现下面这句话则表示运行成功。

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

接下来，打开浏览器，输入 `http://127.0.0.1:5000/`，得到图 10-1 则代表 Flask 环境部署成功。



图 10-1

在上面代码中，`@app.route('/')`表示路由的路径，一般来讲，我们使用 Flask 发布 API 接口，会在此配置相应的路径，以让前端应用程序调用。

10.2 训练 MNIST 模型

在 Flask 环境部署好之后，接下来就要开始正式编写代码了。在正式编写代码之前，先来梳理一下模型的训练过程。

下载数据集

在训练模型之前要下载 MNIST 数据集，相信看过前面章节的读者对 MNIST 数据集一定非常熟悉。MNIST 数据集是由 7 万张图片所构成的，其中包括 6 万张的训练图片和 1 万张的测试图片。下载 MNIST 数据集一般有两种方式：第一种方式是使用 TensorFlow 所提供的 `input_data.py` 文件进行数据集下载；第二种方式是在官网上下

载之后复制到相应的训练目录下。一般来讲，我们使用第一种方式进行数据集下载。

input_data.py 文件代码如下：

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import gzip
import os
import tempfile

import numpy
from six.moves import urllib
from six.moves import xrange
import tensorflow as tf
from tensorflow.contrib.learn.python.learn.datasets.mnist import read_data_sets
```

在这段代码中，第一行代码的意义为绝对引用新特性，使用绝对引入的时候，一般需要指定要引入的顶层 package 名。

第二行代码的含义是导入精确除法，如果我们在程序中没有导入这个特征，“/”符号只能进行整除，也就是取整。只有导入了 division，“/”执行的才是精确除法。

第三行代码的含义是在 Python 2 中可以使用 print('abc')这样的语法，也就是引入 Python 3 的语法，当然，我们本来使用的就是 Python3。

接下来几行代码则是导入下载数据集所必要的库，关于这些库的含义，大家可以通过搜索引擎获取相关的知识。

最后一行代码则表示通过 TensorFlow 的 Datasets 下载 MNIST 数据集。只需要调用此函数，并指定相应的下载路径，就可以下载数据集了。

建立相关模型

下载完数据集之后，接下来就要开始建立数据模型了。在之前的章节中我们学过线性回归的模型和卷积神经网络，那么接下来就分别建立这两个模型，以此对比这两个模型的最终训练结果。在工程中新建一个 model.py 文件，该文件的代码如下：

```

import tensorflow as tf

def regression(x):
    W = tf.Variable(tf.zeros([784, 10]), name="W")
    b = tf.Variable(tf.zeros([10]), name="b")
    y = tf.nn.softmax(tf.matmul(x, W) + b)
    return y, [W, b]

def convolutional(x, keep_prob):
    def conv2d(x, W):
        return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

    def max_pool_2x2(x):
        return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
padding='SAME')

    def weight_variable(shape):
        initial = tf.truncated_normal(shape, stddev=0.1)
        return tf.Variable(initial)

    def bias_variable(shape):
        initial = tf.constant(0.1, shape=shape)
        return tf.Variable(initial)

    x_image = tf.reshape(x, [-1, 28, 28, 1])
    W_conv1 = weight_variable([5, 5, 1, 32])
    b_conv1 = bias_variable([32])
    h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
    h_pool1 = max_pool_2x2(h_conv1)

    W_conv2 = weight_variable([5, 5, 32, 64])
    b_conv2 = bias_variable([64])
    h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
    h_pool2 = max_pool_2x2(h_conv2)

    W_fc1 = weight_variable([7 * 7 * 64, 1024])
    b_fc1 = bias_variable([1024])
    h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
    h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

    h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

    W_fc2 = weight_variable([1024, 10])

```



```
b_fc2 = bias_variable([10])
y = tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
return y, [W_conv1, b_conv1, W_conv2, b_conv2, W_fc1, b_fc1, W_fc2, b_fc2]
```

在上述代码中定义了两个模型函数，分别是 `regression()` 函数和 `convolutional()` 函数。其中 `regression()` 函数所建立的是线性回归模型，而 `convolutional()` 函数所建立的是一个三层卷积神经网络模型。

在线性回归模型中，我们只是简单地定义了权重 W 和偏置量 b ，并使用 TensorFlow 自带的 `tf.nn.softmax()` 函数做了一个 $y=Wx+b$ 的操作，最后分别返回 y , $[W,b]$ 参数。

而在卷积神经网络模型中，分别定义了卷积函数 `conv2d()`、池化函数 `max_pool_2x2()`，通过函数名可以很容易地看出，我们使用的是一个大小为 2×2 的卷积核，以及使用权重函数 `weight_variable()` 和偏置量函数 `bias_variable()` 共同进行网络模型的定义。

紧接着，又通过卷积神经网络做了 2 层卷积，并使用全连接层和池化层进行了一系列相关的处理，最后添加了一个 Softmax 层进行分类选择。

由于前面讲过卷积神经网络训练 MNIST 数据集的例子，所以在这里并没有针对代码中的方法进行过多的分析，如果大家有疑问可以翻阅前面的章节或通过搜索引擎补充相关知识。

使用线性回归模型进行训练

到目前为止，我们已经下载了数据集，并定义了相关的模型，接下来就要使用线性回归模型进行相应的训练：

- (1) 导入数据集。
- (2) 加载模型。
- (3) 创建训练方法和参数。
- (4) 开始训练。
- (5) 保存训练后的模型。

首先导入相关的库和文件：

```
import os
import model
import tensorflow as tf
import input_data
```

这里的 `model` 和 `input_data` 都是刚刚所创建的文件。

然后导入数据并加载模型：

```
data = input_data.read_data_sets('MNIST_data', one_hot=True)

with tf.variable_scope("regression"):
    x = tf.placeholder(tf.float32, [None, 784])
    y, variables = model.regression(x)
```

这里使用了 `tf.variable_scope()` 方法，此方法可以通过设置 `reuse` 标志及初始化来影响域下的变量，在这个方法里，声明了一个变量 `x`，并为其设定了一个形状为 `[None, 784]` 且格式为 `tf.float32` 的张量，还定义了变量 `y` 和 `variables`。这两个变量的值是通过 `model` 文件中的 `regression` 函数返回的。

接下来开始定义训练模型，一般会为训练模型指定交叉熵，也会指定训练所使用的优化器、预测的组数及准确率。

```
y_ = tf.placeholder("float", [None, 10])
cross_entropy = -tf.reduce_sum(y_ * tf.log(y))
train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

在这里，我们使用了一种梯度下降方法 `tf.train.GradientDescentOptimizer()` 作为优化器，并设置其步长为 0.01，使用 `tf.reduce_mean()` 函数来求最后预测结果的平均值。

最后将这些参数使用 `tf.train.Saver` 函数保存，并开始训练。

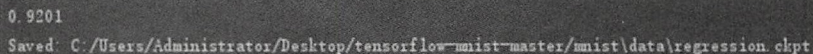
```
saver = tf.train.Saver(variables)
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for _ in range(1000):
```

```
batch_xs, batch_ys = data.train.next_batch(100)
sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})

print(sess.run(accuracy, feed_dict={x: data.test.images, y_:
data.test.labels}))

path = saver.save(
    sess, os.path.join(os.path.dirname(__file__), 'data', 'regression.ckpt'),
    write_meta_graph=False, write_state=False)
print("Saved:", path)
```

训练完成后，将最终的训练结果保存到 regression.ckpt 模型文件中，以供后面开发接口时使用。运行上述代码，可以得到如图 10-2 所示的结果。



```
0.9201
Saved: C:/Users/Administrator/Desktop/tensorflow-mnist-master/mnist\data\regression.ckpt
```

图 10-2

通过图 10-2 可以看到，训练的准确率为 0.9201，并将模型保存到相应的目录下。

使用卷积神经网络进行训练

前面我们使用线性回归训练了 MNIST 数据集，接下来再使用卷积神经网络来训练一次。

使用卷积神经网络进行训练的方式、步骤实际上和使用线性回归的训练方式基本是一样的，即导入数据集、加载模型、创建训练方法和参数、开始训练、保存训练后的模型。为了使训练更加精确，在这里我们将训练的次数调到 20 000 次。代码如下：

```
import os
import model
import tensorflow as tf
import input_data

# from tensorflow.examples.tutorials.mnist import input_data
data = input_data.read_data_sets('MNIST_data', one_hot=True)
# input_data.read_data_sets('MNIST_data', one_hot=True)
# model
with tf.variable_scope("convolutional"):
```



```

x = tf.placeholder(tf.float32, [None, 784], name='x')
keep_prob = tf.placeholder(tf.float32)
y, variables = model.convolutional(x, keep_prob)

# train
y_ = tf.placeholder(tf.float32, [None, 10], name='y')
cross_entropy = -tf.reduce_sum(y_ * tf.log(y))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

saver = tf.train.Saver(variables)

with tf.Session() as sess:
    merged_summary_op = tf.summary.merge_all()
    summary_writer = tf.summary.FileWriter('/tmp/mnist_logs/1', sess.graph)
    summary_writer.add_graph(sess.graph)
    sess.run(tf.global_variables_initializer())
    for i in range(20000):
        batch = data.train.next_batch(50)
        if i % 100 == 0:
            train_accuracy = accuracy.eval(feed_dict={x: batch[0], y_: batch[1],
keep_prob: 1.0})
            print("step %d, training accuracy %g" % (i, train_accuracy))
            sess.run(train_step, feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

    print(sess.run(accuracy, feed_dict={x: data.test.images, y_: data.test.labels,
keep_prob: 1.0}))

    path = saver.save(
        sess, os.path.join(os.path.dirname(__file__), 'data',
'convolutional.ckpt'),
        write_meta_graph=False, write_state=False)
    print("Saved:", path)

```

至此，训练模型这个阶段就可以告一段落了，接下来就可以通过编写 Flask 接口的形式进行模型的调用了。

Flask 接口开发

要将模型发布成 API 接口，先将模型中的参数及模型本身导出。值得注意的是，

TensorFlow 已经封装好了调用模型的接口，在 Python 中可以使用 `saver.restore()` 函数将训练好的 `ckpt` 文件还原出来，并向其接口传递参数。`tf.train.Saver` 类提供了保存和恢复模型的方法，`tf.train.Saver` 构造函数针对图中所有变量或指定列表的变量将 `save` 和 `restore` op 添加到图中。`Saver` 对象提供了运行这些 op 的方法，指定了写入或读取检查点文件的路径，将恢复在模型中定义的所有变量。

所以，当训练完模型之后，就从 Flask 框架的一个 Python 文件中将模型拿出来，然后把所需要的参数传递进去，可以使用如下代码：

```
x = tf.placeholder("float", [None, 784])
sess = tf.Session()

with tf.variable_scope("regression"):
    y1, variables = model.regression(x)
    saver = tf.train.Saver(variables)
    saver.restore(sess, "mnist/data/regression.ckpt")

with tf.variable_scope("convolutional"):
    keep_prob = tf.placeholder("float")
    y2, variables = model.convolutional(x, keep_prob)
    saver = tf.train.Saver(variables)
    saver.restore(sess, "mnist/data/convolutional.ckpt")
```

`x` 是前端所传递进来的参数，在本例中，实际上是通过前端 HTML 页面的 Canvas 画布所生成的图像，再转换成数组传递进来的，具体转换的方式在后面会讲到。

得到 `x` 之后，会将 `x` 分别传入到线性模型和卷积模型中，再调用其 `model` 文件中的相应函数，得到 `y` 值和各个 `variable` 值，其中的 `y` 值在后面进行数据验证时会和传入的参数一起作为新的参数代入验证，而 `variable` 值会作为恢复模型时所加载的训练参数传入。其中线性模型和卷积模型的差别在于：传递卷积模型的时候，还要向其传递一个参数 `keep_prob`，我们知道，`dropout` 函数为了防止过拟合现象的发生，会随机地丢掉一些样本数据，这个参数实际上是指使用 `dropout` 函数时神经元被选中的概率。

最后，定义一个变量 `saver`，并将模型中的变量传递进去，然后通过 `saver.restore()` 函数将模型恢复出来。

当模型恢复出来之后，就可以定义两个函数，将前端输入的值传递进来。

```
def regression(input):  
    return sess.run(y1, feed_dict={x: input}).flatten().tolist()  
  
def convolutional(input):  
return sess.run(y2, feed_dict={x: input, keep_prob: 1.0}).flatten().tolist()
```

其中，y1 是线性模型的参数，y2 是卷积模型的参数。

当函数计算全部定义完成后，就可以编写一个 Flask 接口，将前端所传递的值传入，并通过调用模型得到所需要的返回值，然后统一打包成 JSON 字符串格式，返回给前端，具体代码如下：

```
@app.route('/api/mnist', methods=['post'])  
def mnist():  
    input = ((255 - np.array(request.json, dtype=np.uint8)) / 255.0).reshape(1, 784)  
    output1 = regression(input)  
    output2 = convolutional(input)  
    return jsonify(results=[output1, output2])  
  
@app.route('/')  
def main():  
    return render_template('index.html')  
  
if __name__ == '__main__':  
    app.debug=True  
    app.run(host='0.0.0.0',port=8000)
```

首先定义了一个接口 '/api/mnist'，并为其指明其接口传递方式为 post 方式；再将传递进来的值 request.json 统一处理（reshape）成模型所需要的格式，并将其传递到模型中；最后将所得到的两个模型的结果组合成一个 JSON 数组返回给前端。前端接到 JSON 数组后，就可以将其解析并显示到界面上。

最后，我们指明其前端界面为 index.html 文件，并定义运行接口为 8000 接口。至此，整个接口部分就开发完成，接下来就可以写一个前端界面进行调用了。

前端界面编写和调用

前面完成了模型的训练和接口的开发，接下来要做的就是结合前端页面进行模型的调用，调用这个接口，获取最后的预测值并显示到界面上，如图 10-3 所示。

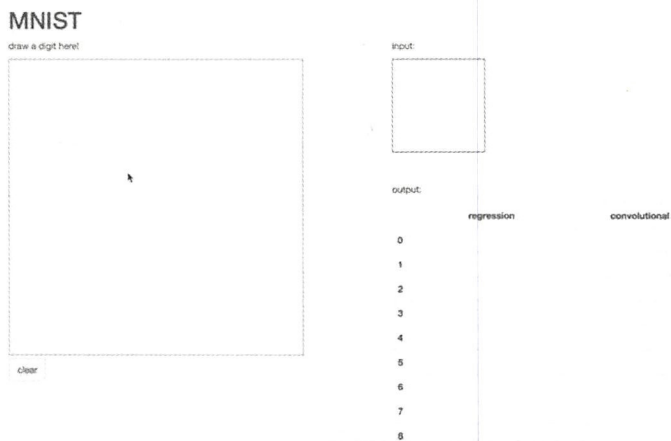


图 10-3

在这个界面中，左侧实际上是一个 Canvas 画布，可以使用鼠标来绘制手写数字；右上角是 input 部分，显示左侧所绘制图像的缩略图，右下角的 output 部分是针对所绘制的数字而对应的 0~9 的概率值，如图 10-4 所示。

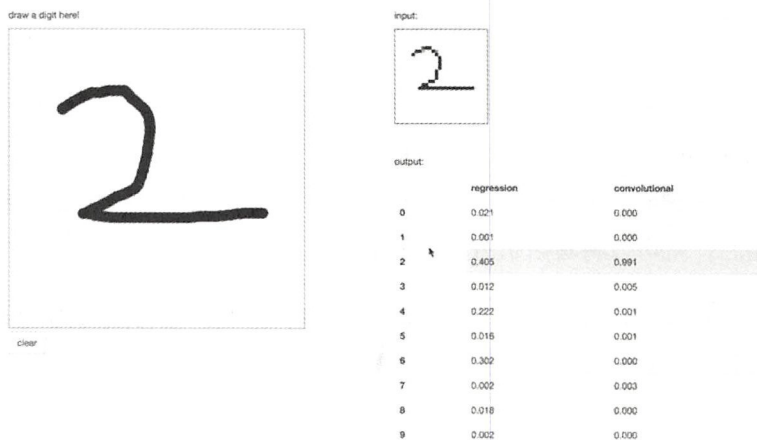


图 10-4

例如我们在图 10-4 左侧绘制了一个数字 2，这个时候分别通过线性模型和卷积模型进行预测。通过线性模型预测认为，绘制的图像是 2 的概率为 40.5%；通过卷积模型预测认为，绘制的图像是 2 的概率为 99.1%。这说明我们所训练的准确率还是比较高的。

那么具体是如何实现的呢？

首先写一个简单的 HTML 页面：index.html。

```
<!DOCTYPE html>
<html>
  <head>
    <title>MNIST</title>
    <link rel="stylesheet" href="{{ url_for('static',
filename='css/bootstrap.min.css') }}">
    <script type="text/javascript" src="{{ url_for('static',
filename='js/jquery.min.js') }}"></script>
    <script type="text/javascript" src="{{ url_for('static',
filename='js/main.js') }}"></script>
  </head>
  <body>
    <div class="container">
      <h1>MNIST</h1>
      <div class="row">
        <div class="col-md-6">
          <p>draw a digit here!</p>
          <canvas id="main"></canvas>
          <p>
            <button id="clear" class="btn btn-default">clear</button>
          </p>
        </div>
        <div class="col-md-6">
          <p>input:</p>
          <canvas id="input" style="border:1px solid" width="140"
height="140"></canvas>
          <hr>
          <p>output:</p>
          <table id="output" class="table">
            <tr>
              <th class="col-md-1"></th>
              <th class="col-md-2">regression</th>
            </tr>
          </table>
        </div>
      </div>
    </div>
  </body>
</html>
```



```
<th class="col-md-2">convolutional</th>
</tr>
<tr>
  <th>0</th>
  <td></td>
  <td></td>
</tr>
<tr>
  <th>1</th>
  <td></td>
  <td></td>
</tr>
<tr>
  <th>2</th>
  <td></td>
  <td></td>
</tr>
<tr>
  <th>3</th>
  <td></td>
  <td></td>
</tr>
<tr>
  <th>4</th>
  <td></td>
  <td></td>
</tr>
<tr>
  <th>5</th>
  <td></td>
  <td></td>
</tr>
<tr>
  <th>6</th>
  <td></td>
  <td></td>
</tr>
<tr>
  <th>7</th>
  <td></td>
  <td></td>
</tr>
<tr>
```

```

        <th>8</th>
        <td></td>
        <td></td>
    </tr>
    <tr>
        <th>9</th>
        <td></td>
        <td></td>
    </tr>
</table>
</div>
</div>
</div>
</body>
</html>

```

这个界面看起来代码量比较大，实际上非常简单，如图 10-3 所示。在左侧建立了一个 Canvas 画布，`<canvas id="main"></canvas>`，在右上角建立了一个用于显示所绘制图像的 Canvas 画布，`<canvas id="input" style="border:1px solid" width="140" height="140"></canvas>`，最后在下方放置一个 table 用来显示所得到的结果。

在编写完 HTML 界面后，紧接着为其编写 JS 文件：main.js。

```

class Main {
    constructor() {
        this.canvas = document.getElementById('main');
        this.input = document.getElementById('input');
        this.canvas.width = 449; // 16 * 28 + 1
        this.canvas.height = 449; // 16 * 28 + 1
        this.ctx = this.canvas.getContext('2d');
        this.canvas.addEventListener('mousedown', this.onMouseDown.bind(this));
        this.canvas.addEventListener('mouseup', this.onMouseUp.bind(this));
        this.canvas.addEventListener('mousemove', this.onMouseMove.bind(this));
        this.initialize();
    }
    initialize() {
        this.ctx.fillStyle = '#FFFFFF';
        this.ctx.fillRect(0, 0, 449, 449);
        this.ctx.lineWidth = 1;
        this.ctx.strokeRect(0, 0, 449, 449);
        this.ctx.lineWidth = 0.05;
    }
}

```



```
for (var i = 0; i < 27; i++) {
  this.ctx.beginPath();
  this.ctx.moveTo((i + 1) * 16, 0);
  this.ctx.lineTo((i + 1) * 16, 449);
  this.ctx.closePath();
  this.ctx.stroke();

  this.ctx.beginPath();
  this.ctx.moveTo(0, (i + 1) * 16);
  this.ctx.lineTo(449, (i + 1) * 16);
  this.ctx.closePath();
  this.ctx.stroke();
}
this.drawInput();
$('#output td').text('').removeClass('success');
}
onMouseDown(e) {
  this.canvas.style.cursor = 'default';
  this.drawing = true;
  this.prev = this.getPosition(e.clientX, e.clientY);
}
onMouseUp() {
  this.drawing = false;
  this.drawInput();
}
onMouseMove(e) {
  if (this.drawing) {
    var curr = this.getPosition(e.clientX, e.clientY);
    this.ctx.lineWidth = 16;
    this.ctx.lineCap = 'round';
    this.ctx.beginPath();
    this.ctx.moveTo(this.prev.x, this.prev.y);
    this.ctx.lineTo(curr.x, curr.y);
    this.ctx.stroke();
    this.ctx.closePath();
    this.prev = curr;
  }
}
getPosition(clientX, clientY) {
  var rect = this.canvas.getBoundingClientRect();
  return {
    x: clientX - rect.left,
    y: clientY - rect.top
  }
}
```

```

    };
}
drawInput() {
    var ctx = this.input.getContext('2d');
    var img = new Image();
    img.onload = () => {
        var inputs = [];
        var small = document.createElement('canvas').getContext('2d');
        small.drawImage(img, 0, 0, img.width, img.height, 0, 0, 28, 28);
        var data = small.getImageData(0, 0, 28, 28).data;
        for (var i = 0; i < 28; i++) {
            for (var j = 0; j < 28; j++) {
                var n = 4 * (i * 28 + j);
                inputs[i * 28 + j] = (data[n + 0] + data[n + 1] + data[n + 2]) / 3;
                ctx.fillStyle = 'rgb(' + [data[n + 0], data[n + 1], data[n + 2]].join(',') + ')';
                ctx.fillRect(j * 5, i * 5, 5, 5);
            }
        }
        if (Math.min(...inputs) === 255) {
            return;
        }

        $.ajax({
            url: '/api/mnist',
            type: 'post',
            contentType: 'application/json',
            data: JSON.stringify(inputs),
            success: (data) => {
                for (let i = 0; i < 2; i++) {
                    var max = 0;
                    var max_index = 0;
                    for (let j = 0; j < 10; j++) {
                        var value = Math.round(data.results[i][j] * 1000);
                        if (value > max) {
                            max = value;
                            max_index = j;
                        }
                    }
                    var digits = String(value).length;
                    for (var k = 0; k < 3 - digits; k++) {
                        value = '0' + value;
                    }
                    var text = '0.' + value;
                }
            }
        });
    };
}

```



```

        if (value > 999) {
            text = '1.000';
        }
        $('#output tr').eq(j + 1).find('td').eq(i).text(text);
    }
    for (let j = 0; j < 10; j++) {
        if (j === max_index) {
            $('#output tr').eq(j + 1).find('td').eq(i).addClass
('success');
        } else {
            $('#output tr').eq(j + 1).find('td').eq(i).removeClass
('success');
        }
    }
    }
    });
    };
    img.src = this.canvas.toDataURL();
}
}

$(() => {
    var main = new Main();
    $('#clear').click(() => {
        main.initialize();
    });
});
});

```

上述代码大体上可以分为 3 大部分：第一部分是对于绘制画布的处理，通过监听鼠标的按下、移动和抬起来绘制鼠标的轨迹，从而形成所绘制的数字；第二部分是 will 绘制的图像转换成模型所需要的数组格式；第三部分是将转换好的格式，通过 AJAX 的方式调用 `/api/mnist` 接口，将数据打包成 JSON 格式并以 POST 的形式传递给后端模型，然后将模型所返回的 JSON 数组进行解析，并显示在界面上。

在这里，`main.py` 文件的接口接收前端所传递的数据之后，就会调用线性和卷积的模型，并将输入数据传递进去，待模型返回计算结果后，再组合成 JSON 数组返回给前端进行显示。程序运行后，进行绘制，运行出来的结果就会如图 10-4 所示。

10.3 小结

本章只是以 MNIST 模型作为一个简单的示例结合 Flask 进行模型接口发布,读者在实际应用的过程中,可以根据真实的用户需求,将任意模型以这种形式进行接口发布,从而使任何前端界面(包括 PC 端和移动端等)可以调用。

第 11 章

TensorFlow 模型的发布与部署

第 10 章讲过如何使用 TensorFlow 结合 Python 的 Flask 框架进行 RESTful 形式的模型发布,通过这种方式可以很容易地将一个 TensorFlow 的模型发布成一个 RESTful API,供其他开发人员调用。这种部署方式虽然简单便捷,但是也存在着一一定的局限性。

首先,使用 Flask 框架进行部署的时候,在工程内部调用 TensorFlow 模型,必须要使用 Python 语言进行接口的发布,而不能使用其他语言,限制颇多;其次,我们使用 Flask 框架的形式发布成 API,其实相当于在本身模型接口的基础上再增加了一个接口,外面进行了一层包装,这样做的效率和响应速度相对于直接调用 TensorFlow 模型来讲肯定会慢得多。那么,如何才能直接调用 TensorFlow 训练出来的模型呢?其实,谷歌已经为我们提供了 TensorFlow Serving。采用这种 TensorFlow 原生的方式进行模型发布,就可以跳过中间 Flask 接口层,直接将 TensorFlow 模型的接口发布出来,让其他的应用程序进行调用。

11.1 TensorFlow Serving 的前导知识

TensorFlow Serving 是一款灵活的高性能机器学习服务系统,专为生产环境而设计。TensorFlow Serving 可以轻松部署新算法和实验,同时保持相同的服务器体系结

构和 API。TensorFlow Serving 提供与 TensorFlow 模型的即时可用集成，且可以轻松扩展，以服务其他类型的模型和数据。

在详细了解什么是 TensorFlow Serving 之前，先来了解一下关于 TensorFlow Serving 的前导知识。

Servables

Servables 是 TensorFlow Serving 中的中心抽象，也是客户用来执行计算的基础对象，例如查找计算或者推理计算等。Servables 的大小和粒度相对比较灵活，一般来讲，单个的 Servables 可能包含从查找表的单个分片到单个模型，再到推理模型元组中的任何内容。Servables 的表现形式可以是任何类型的接口。典型的 Servables 包括 TensorFlow SavedModelBundle (`tensorflow::Session`) 和用于嵌入或查找词汇的查找表。

Servables Versions

TensorFlow Serving 可以处理一个或多个在 Servables 的生命周期内的单个服务器实例的版本，这就使得新的算法配置、权重和其他的数据能够随时加载，Servables Versions 也允许多个版本的 Servables 被同时加载，并支持逐步开展实验。在服务期间，客户端也可以为特定的模型请求最新版本或者指定的版本。

Servables Streams

一个 Servables Streams 是 Servables 的版本序列，一般按照增长的版本号进行排序。

Models

TensorFlow Serving 可以将模型表示为一个或多个 Servables，一个机器学习模型可以包括一个或多个算法（包括权重学习）、查找表和嵌入表等，也可以将复合模型表示成多个独立的服务或单个 Servable 服务，同时，Servables 也可以是模型的一部分。例如，一个大的查找表可以在许多 TensorFlow Serving 实例中被分割开来。

Loaders

Loaders 被用来管理一个 Servable 的生命周期，Loader API 能够使通用基础架构独立于特定的学习算法、数据或涉及的相关产品使用案例之外。也可以这样说，Loaders 对加载和卸载一个 Servable 进行了标准化管理。

Sources

Sources 是用于查找和提供服务的插件模块，每一个 Sources 都提供了 0 个或者多个 Servables Streams。对于每一个 Servables Stream，Sources 都可以为其提供一个可供加载版本的 Loader 实例。一般来讲，一个 Sources 实际是与 0 个或多个 SourceAdapter 链接在一起的，并且在链中的最后一个项目会发出 Loaders。

在 TensorFlow Serving 中的 Sources 接口可以从任意一台存储系统中发现 Servable 服务。TensorFlow Serving 也包含了通用的 Sources 实现参考。例如，Sources 可以访问诸如 RPC 之类的机制，并且可以对文件系统做轮询操作。

Sources 也可以维护多个 Servables 或版本的状态，对于 Servables 版本增量更新而言，这是非常有用的。

Aspired Versions

Aspired Versions 表示应该加载并准备好的一组 Servables Versions。Sources 为 Servables Stream 一次传送一组 Servables Version。当一个 Source 向 Managers 提供一个新的可用版本列表时，它将取代 Servables Stream 的前一个列表。管理器卸载的任何一个先前加载的版本都不会出现在之后的列表中。

Managers

Managers 用来处理整个 Servables 的生命周期，其中包括：

- (1) 加载 Servables。
- (2) 为 Servables 提供服务。

(3) 卸载 Servables。

Managers 可以监听 Sources 并且追踪所有的版本。Managers 试图完成来自 Sources 的请求，但是如果所需要的资源不可用，则可能会拒绝加载所需要的版本。Managers 也可以推迟程序的卸载，一般等到新版本加载完成后再卸载，因为要确保始终加载至少一个版本。

不过值得庆幸的是，TensorFlow Serving Managers 提供了一个简单的接口 `GetServableHandle()`，用于客户端访问加载的 Servables 实例。

Core

TensorFlow Serving Core 通过 TensorFlow 的 API 来管理 Servables 的生命周期和指标，TensorFlow Serving Core 将 Servables 和 Loaders 视为不透明的对象。

图 11-1 为 Servables 的生命周期示意图。

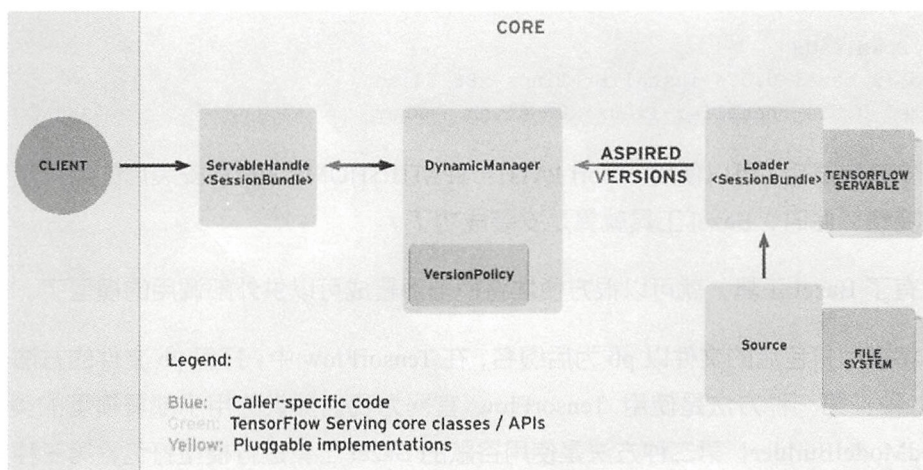


图 11-1

如图 11-1 所示，客户端的功能就是通过 `ServableHandle` 向 Managers 询问 Servables，并请求其版本。一般来讲，要么明确指定版本，要么只是要求最新版本，Managers 在接到请求后，会返回可用版本的句柄，而这个版本是通过 Loaders 所提供的。从图 11-1 中可以看到，Sources 插件会为特定的版本创建一个 Loaders，Loaders 包含加载

Servables 所需要的任何元数据，Sources 使用回调来通知 Managers 的 Aspired 版本。Managers 应用相应的版本策略来确定下一个要执行的动作，比如加载版本，还是卸载之前的版本。如果 Managers 确定它是安全的，则会为 Loaders 提供所需要的资源，并通过 Loaders 加载新的版本，最终返回给客户端。

11.2 TensorFlow Serving 模型打包

在了解完 TensorFlow Serving 的前导知识之后，那么就正式学习如何使用 TensorFlow Serving 来部署一个 TensorFlow 模型。

要使用 TensorFlow Serving 部署模型，则需要一个必备工具——Bazel。Bazel 是一个类似于 Maven 和 Gradle 的开源构建和测试工具，支持多种语言的项目，并且可以为多个平台构建和输出，可以直接从 Bazel 的官方网站上下载最新版本，并使用如下命令进行安装。

```
cd ~/Downloads
chmod +x bazel-0.5.4-installer-linux-x86_64.sh
./bazel-0.5.4-installer-linux-x86_64.sh --user
```

安装完成后，可以使用 `export PATH="$PATH:$HOME/bin"` 将安装后的目录设置为环境变量，此时，Bazel 工具就算是安装成功了。

有了 Bazel 工具，就可以很方便地将模型部署成可以供外部调用的模型了。

将模型打包后的文件以 pb 为后缀名，在 TensorFlow 中，打包 pb 文件的方法一般有 3 种：第一种方法是使用 TensorFlow 官网为我们提供的用于部署模型的类——`SavedModelBuilder`；第二种方法是使用谷歌的 Bazel 工具进行模型打包；第三种方法是使用 TensorFlow 的 `tf.gfile.GFile()` 函数打包。下面分别介绍一下这三种方法。

使用 `SavedModelBuilder` 类打包

TensorFlow 官方为我们提供了用于部署模型的类——`SavedModelBuilder`。`SavedModelBuilder` 类可用来构建 `SavedModel` 协议缓冲区，也可将多个元图（Meta

Graphs) 保存为单一的与语言无关的 `SaveModelBuilder` 的一部分, 同时可以共享变量和资源。

一般来讲, 在构建 `SavedModel` 的时候, 第一个元图必须使用变量保存, 之后的元图将会简单地和其图的定义一起保存。如果资源也需要保存的话, 则需要在添加元图定义的时候提供资源。如果多个元图定义与同名的资源相关联, 则只保留第一个版本。添加到 `SavedModel` 的每一个元图都必须使用标签进行注解, 这些标签提供了一种方法来识别要加载和恢复的特定元图, 以及共享的一组变量和资源。

我们来看一段来自 TensorFlow 官方网站的示例代码:

```
export_dir = 'model/version'
builder = tf.saved_model.builder.SavedModelBuilder(export_dir)
with tf.Session(graph=tf.Graph()) as sess:
    builder.add_meta_graph_and_variables(sess,
                                         ["foo-tag"],
                                         signature_def_map=foo_signatures,
                                         assets_collection=foo_assets)
with tf.Session(graph=tf.Graph()) as sess:
    builder.add_meta_graph(["bar-tag", "baz-tag"])
builder.save()
```

这段代码是 `SavedModelBuilder` 的典型用法, 在这里首先定义了一个模型的导出路径, 再使用 `tf.saved_model.builder.SavedModelBuilder` 函数将路径传入, 即建立一个 `builder`。紧接着在 `session` 运行图部分, 使用 `builder.add_meta_graph_and_variables()` 函数将当前的 `graph` 和 `variables` 添加进去, 该方法的定义如下:

```
add_meta_graph_and_variables(
    sess,
    tags,
    signature_def_map=None,
    assets_collection=None,
    legacy_init_op=None,
    clear_devices=False,
    main_op=None,
    strip_default_attrs=False
)
```

从上面代码可以知道, `add_meta_graph_and_variables()` 函数可以接收 8 个参数。

- **sess**: 保存元图和变量的 TensorFlow 会话。
- **tags**: 用于保存元图的一组标签。
- **signature_def_map**: 签名到元图的映射。
- **assets_collection**: 要与 SavedModel 一起保存的资源的集合。
- **legacy_init_op**: 对操作或操作组的遗留支持, 在加载并恢复 op 后执行。
- **clear_devices**: 如果要清除默认图形上的设备信息, 则设置为 True。
- **main_op**: 加载图形时要执行的操作或操作组。
- **strip_default_attrs**: 布尔值。如果为 True, 则将从 NodeDefs 中删除默认值属性。

一般来讲, 我们在使用的时候只需要指定前面 5 个参数即可。

`builder.add_meta_graph()`函数与 `add_meta_graph_and_variables()`函数类似, 仅仅将图的信息添加进去而已, 具体参数可以参照 TensorFlow 的官方文档。

最后使用 `builder.save()`函数将模型保存下来, 以供后续使用。模型被保存后, 会在保存模型的目录中生成一个 `saved_model.pb` 文件和一个 `variables` 文件夹。其中 `saved_model.pb` 文件就是我们保存下来的模型文件, 也是之后我们要发布时所使用的文件; `variables` 文件夹里存放的就是与模型相关的一系列参数。这两部分会被存到一个自定义的模型目录下, 当发布模型时, 需要对整个文件夹进行操作。

值得注意的是, 在导出模型的时候, 必须将保存导出模型的目录设置为空目录, 否则会出现一个该目录已存在的错误, 所以, 在设置 `export_dir` 值的时候, 务必要保证里面没有任何数据。

使用 Bazel 打包

使用 Bazel 打包, 首先要安装 Bazel, 这里有一个前提, 就是 Bazel 的安装依赖于 Java 的运行环境, 一般要求其 JDK 的版本不低于 1.8。Java 可以在 JDK 官方网站下载, 下载后需要按照相关教程配置 `JAVA_HOME`、`PATH`、`CLASSPATH` 三个环境变量。在配置完 Java 环境后, 就可以安装并使用 Bazel 进行打包了。

Bazel 的使用也非常简单:

```
$> bazel build -c opt // tensorflow_serving / example: mnist_saved_model  
$> bazel-bin / tensorflow_serving / example / mnist_saved_model / tmp / mnist_model
```

第一行的意思是将我们的训练源码进行构建，也就是训练模型的过程；第二行的意思则是将训练好的 TensorFlow 打包成一个模型文件，打包后同样生成 saved_model.pb 文件和 variables 文件夹。

虽然使用 Bazel 打包是谷歌官方推荐的，但是在实际生产过程中并不常用。

使用 TensorFlow 的 tf.gfile.GFile()函数打包

在 TensorFlow 中,还有一种方法比较常用,就是使用 tf.gfile.GFile()函数打包,其实这种方式的主要原理是充分利用了 TensorFlow 中的计算图封装,将计算图的 GraphDef 部分导出,然后使用其图的信息进行从输入到输出的计算。

我们来看一个小例子:

```
import tensorflow as tf  
from tensorflow.python.framework import graph_util  
from tensorflow.python.platform import gfile  
  
if __name__ == "__main__":  
    a = tf.Variable(tf.constant(5., shape=[1]), name="a")  
    b = tf.Variable(tf.constant(6., shape=[1]), name="b")  
    c = a + b  
    init = tf.initialize_all_variables()  
    sess = tf.Session()  
    sess.run(init)  
    # 导出当前计算图的 GraphDef 部分  
    graph_def = tf.get_default_graph().as_graph_def()  
    # 保存指定的节点,并将节点值保存为常数  
    output_graph_def = graph_util.convert_variables_to_constants(sess, graph_def,  
        ['add'])  
    # 将计算图写入到模型文件中  
    with tf.gfile.GFile("models.pb", "wb") as f:  
        f.write(output_graph_def.SerializeToString())
```

上面一段代码就是使用 tf.gfile.GFile()函数打包,将图的信息进行保存,然后写入 pb 文件中。

11.3 TensorFlow Serving 模型的部署和调用

pb 文件就是我们所打包出来的模型，有了 pb 文件后，就可以使用 Python 或者其他平台对其进行调用。下面来看看如何使用 Python 调用 pb 文件。

在 Python 中，想要调用 pb 文件，先从 pb 文件中把图的相关信息取出来：

```
output_graph_def = tf.GraphDef()
output_graph_path = '../model/expert-graph.pb'
with open(output_graph_path, 'rb') as f:
    output_graph_def.ParseFromString(f.read())
y = tf.import_graph_def(output_graph_def, name="")
```

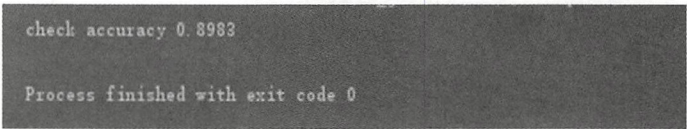
取出图的信息之后，创建一个 session 并定义相关的输入和输出参数：

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    input = sess.graph.get_tensor_by_name("input:0")
    output = sess.graph.get_tensor_by_name("output:0")
```

接下来，输入数据，并得到输出的准确率，在这里使用 MNIST 测试数据集进行测试：

```
y_conv_2 = sess.run(output, feed_dict={input: mnist.test.images})
y_2 = mnist.test.labels
correct_prediction_2 = tf.equal(tf.argmax(y_conv_2, 1), tf.argmax(y_2, 1))
accuracy_2 = tf.reduce_mean(tf.cast(correct_prediction_2, "float"))
print("check accuracy %g" % sess.run(accuracy_2))
```

最后，打印测试结果如图 11-2 所示。



```
check accuracy 0.8983

Process finished with exit code 0
```

图 11-2

第 12 章

TensorFlow Lite 牛刀小试

在前面的章节中，我们可以使用 TensorFlow 训练模型进行预测，并将其打包成一个可供其他程序调用的接口。谷歌还为我们提供了一套全新的针对移动设备而设计的调用方法——TensorFlow Lite。

12.1 什么是 TensorFlow Lite

TensorFlow Lite 是 TensorFlow 针对移动设备和嵌入式设备而推出的一套轻量级的解决方案，它支持在设备上运行更小的二进制文件，且具有更低的延迟。另外，TensorFlow Lite 还支持在 Android 上神经网络 API 的硬件加速，Android 从 8.1 版本开始提供了专门针对 TensorFlow 的 API 接口，能够快速实现 Android 与 TensorFlow 的结合。另外，TensorFlow Lite 使用了很多技术来实现低延迟，比如优化了移动应用程序的内核，预融合激活，以及允许更小、更快的模型来量化内核。另外，在 iOS 端也可以使用 TensorFlow Lite 进行很好的模型调用。

TensorFlow Lite 支持一系列量化和浮点的核心运算符，这些运算符已经针对移动平台进行了调整。它们将预融合激活和偏差结合，进一步提高性能和量化精度。此外，TensorFlow Lite 也支持在模型中自定义操作。

TensorFlow Lite 定义了基于 FlatBuffers 的新模型文件格式，这个格式就是在移动设备中所用到的 tflite 格式。而 FlatBuffers 是一个开源的跨平台序列化库，其主要优点在于，它所占用的空间相较于其他协议缓冲区而言会更小。在大多 Android 设备上，我们都可以使用硬件进行物理加速，而 TensorFlow Lite 的一大特性就是提供了一个接口来使用硬件加速，当然，前提是硬件能够支持这种加速方式。目前深度学习领域正在逐步地改变计算模式，很多移动和嵌入式设备的崛起也使得深度学习在移动领域中有了新的机遇，尤其是在移动手机上的相机和语音交互的推动下，消费者对于移动设备的期望值也变得越来越高的，希望移动设备能够带来近似于人类交互的人性化体验，因此产生了很多在移动设备上使用模型的需求。

目前，谷歌已经提供了一些可以直接在移动设备上使用的模型，并已实现“开箱即用”。这些模型包括但不限于：

- **Inception-V3 模型**：一种流行的模型，用于图像检测和目标识别。通过这个模型可以利用手机上的摄像头进行物体识别。
- **MobileNets 模型**：一系列计算机视觉模型，旨在最大程度地有效提高精确度，同时也考虑到对移动设备或嵌入式应用的资源限制问题。其优点是功耗低、速度快、资源更小，但是正因如此，也暴露出其缺点，即精确度不如 Inception-V3 模型。

TensorFlow Lite 的整体架构如图 12-1 所示。我们在使用 TensorFlow Lite 时，首先需要准备一个已经训练好的模型，这个训练好的模型可以是 ckpt 文件，或者是 pb 文件；然后，使用 TensorFlow 所提供的转换器将我们所训练好的模型转化成 TensorFlow Lite 所独有的格式（.tflite）；最后，就可以在移动应用程序中使用转换后的 tflite 文件进行模型的调用。

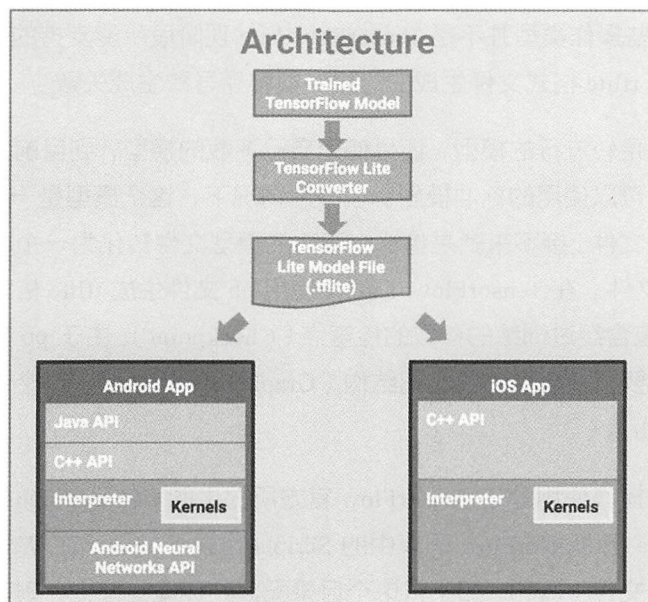


图 12-1

12.2 如何使用 TensorFlow Lite 模型

要想在移动设备中使用 TensorFlow Lite 模型进行预测，第一步就是需要一个 TensorFlow Lite 格式的文件——.tflite 格式文件（也可以直接使用 pb 文件进行预测，不过 pb 文件体积较大，一般不建议直接在移动设备中使用），然后将这个 tflite 文件导入 Android 或 iOS 的 APP 应用中进行调用预测。

那么如何生成 tflite 文件呢？首先需要有一个已训练好的 pb 格式的模型文件。

模型一般有两种来源：第一种来源就是使用现阶段比较流行的模型（例如 Inception-V3 模型），然后使用自己自定义的数据集重新训练这些模型，再将模型转换为 tflite 格式在移动设备上调用；第二种来源就是使用自己训练的模型，再将这个模型转换成 tflite 格式在移动设备上进行调用。一般来讲，我们建议使用第一种进行操作，因为 TensorFlow Lite 目前推出时间较短，仅支持部分 op（操作类型），而在很多自定义模型（尤其是和自然语言处理、RNN 相关的模型）中，会用到更多不常用的

操作类型，而这些操作类型并不是 TensorFlow Lite 现阶段所能支持的。因此使用自己训练的模型进行 tflite 格式文件生成的时候，很容易导致生成失败。

无论使用的是较流行的模型，还是使用自己下载的模型，到目前为止，我们至少已经拥有了一个可以使用的标准模型。在一般情况下，这个模型是一个以 pb 或 ckpt 为后缀名的模型文件。接下来就是将这个已有的模型文件转化为一个可以在移动设备上使用的 tflite 文件。在 TensorFlow Lite 中使用 pb 文件生成 tflite 格式的文件的前提是 pb 文件必须包含经过训练的参数的检查点（Checkpoint）。由于 pb 文件一般只包含图的结构，所以要将检查点的值和图结构（GraphDef）相结合，这个过程被称为冻结图（Freeze Graph）。

冻结图这一操作使用的是 TensorFlow 官方所提供的 freeze_graph 工具，这个工具可以在 Anaconda 下 TensorFlow 环境中的 Scripts 目录下找到，在 Windows 系统中，它是一个可执行文件（exe）。如果将这个目录配置到环境变量中，则可在以后的操作中直接使用 freeze_graph 命令，无须输入全路径。具体代码如下：

```
freeze_graph --input_graph=/tmp/mobilenet_v1_224.pb \  
--input_checkpoint=/tmp/checkpoints/mobilenet-10202.ckpt \  
--input_binary=true \  
--output_graph=/tmp/frozen_mobilenet_v1_224.pb \  
--output_node_names=MobileNetV1/Predictions/Reshape_1
```

freeze_graph 命令有 5 个参数：

- **input_graph**：输入的模型文件，一般是使用打包好的 pb 文件；
 - **input_checkpoint**：输入的 ckpt 文件，一般是在训练结束后需要保存相应的 ckpt 文件；
 - **input_binary**：输入的是否是二进制数，一般要求必须启用 input_binary 标志，以便 protobuf 以二进制格式读取和写入；
 - **output_graph**：输出的图，一般指定绑定好参数的图模型文件；
 - **output_node_names**：输出的节点名称，一般来讲，可以在 TensorBoard 中找到它。
- 通过执行上述命令，我们就可以将模型文件进行冻结图操作，冻结图实际上是一个将 pb 文件转换成 tflite 文件过程中的中间步骤。

注意：在 TensorFlow1.8 版本以前的操作，只有冻结图才能转换成 tflite 文件，但是在 1.8 版本之后，实际上这个步骤是可以省略的。

谷歌在 TensorFlow 中为我们提供了一个 Tensorflow Optimizing Converter(TOCO) 工具，利用 TOCO 工具，可以直接将冻结图转换成 Android 可以使用的 tflite 文件。如果使用 Anaconda 作为开发环境，则 TOCO 工具的位置与 freeze_graph 位于同一个目录。可以使用以下命令来生成 tflite 文件：

```
toco --input_file=$(pwd)/mobilenet_v1_1.0_224/frozen_graph.pb \
--input_format=TENSORFLOW_GRAPHDEF \
--output_format=TFLITE \
--output_file=/tmp/mobilenet_v1_1.0_224.tflite \
--inference_type=FLOAT \
--input_type=FLOAT \
--input_arrays=input \
--output_arrays=MobilenetV1/Predictions/Reshape_1 \
--input_shapes=1,224,224,3
```

先解释一下这个命令中的几个参数：

- **input_file**：输入模型文件，其中\$(pwd)/mobilenet_v1_1.0_224/是指模型所在的路径，frozen_graph.pb 是输入的冻结后的模型文件，也就是在使用 freeze_graph 命令时的 output_graph；
- **input_format**：将输入的模型转换为 TensorFlow 可以使用的模型，所以这里的参数一般固定为 TENSORFLOW_GRAPHDEF；
- **output_format**：将输出格式转化为 tflite 格式；
- **output_file**：设置输出的 tflite 文件路径和名称；
- **inference_type**：接口的数据类型，如果没有量化模型，那么默认为 float 类型；
- **input_type**：输入数据类型，同上；
- **input_arrays、output_array、input_shape**：这三个参数一般不会明显地体现出来，可以在 TensorBoard 中找到；
- **input_shapes**：输入参数的 Tensor 大小。

通过执行以上命令，就可以生成相应的 tflite 文件，然后将这个 tflite 文件作为外部资源加载到 Android 程序开发环境中，就可以进行接口调用。

上述方式是通过 TensorFlow Lite 所提供的外部工具进行打包的。除此之外，TensorFlow 还提供了另外一种生成 tflite 文件的方式，即通过代码生成 tflite 文件。前面讲过可以通过将图的信息写入 pb 文件的方式进行模型的打包，同样地，在 TensorFlow Lite 中也支持这种方式，我们可以很方便地代码打包，不过这种方式是在 TensorFlow 1.7 版本以上才支持的。通过一个小例子来说明如何代码打包。

```
import tensorflow as tf

img = tf.placeholder(name="img", dtype=tf.float32, shape=(1, 64, 64, 3))
val = img + tf.constant([1., 2., 3.]) + tf.constant([1., 4., 4.])
out = tf.identity(val, name="out")

with tf.Session() as sess:
    tflite_model = tf.contrib.lite.toco_convert(sess.graph_def, [img], [out])
    open("converted_model.tflite", "wb").write(tflite_model)
```

上面这段代码所实现的功能非常简单，首先声明了一个 tf.float32 类型的变量 img，并将其 shape 设置为(1, 64, 64, 3)；其次再定义一个变量，让 img 和另外两个常量相加得到一个输出值，并把输出值赋值给 out 变量；然后通过 tf.contrib.lite.toco_convert() 函数传入运行图、输入参数、输出参数；最后生成一个 converted_model.tflite 文件。随着 TensorFlow 版本的更新，建议逐步使用代码来替代工具生成 tflite 文件。

12.3 TensorFlow Lite 与 Android 结合实现图像识别

利用谷歌提供的官方例子来讲解一下 TensorFlow Lite 如何与 Android 结合进行图像识别，这里所用的模型为 MobileNet 模型。

首先在 TensorFlow 官方的 GitHub 上下载 Android 的 Demo 源码。

TensorFlow Lite 对 Android 环境的要求如下：

- Android Studio 的版本大于等于 3.0；
- Android SDK 的 target 版本大于等于 25；

- Android NDK 的版本大于等于 14。

在确保符合上述条件的前提下，导入项目，在 Android Studio 单击 File→open 菜单导入文件，分别如图 12-2、图 12-3 所示。

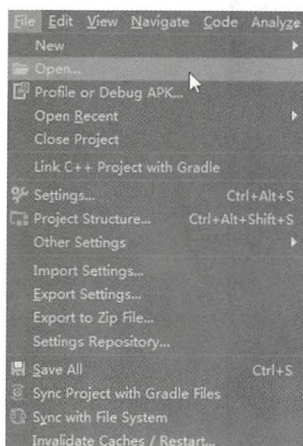


图 12-2

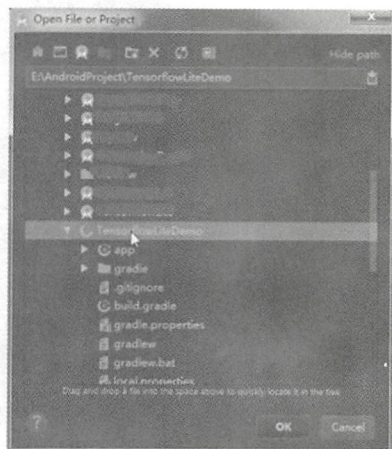


图 12-3

项目导入后，界面如图 12-4 所示。

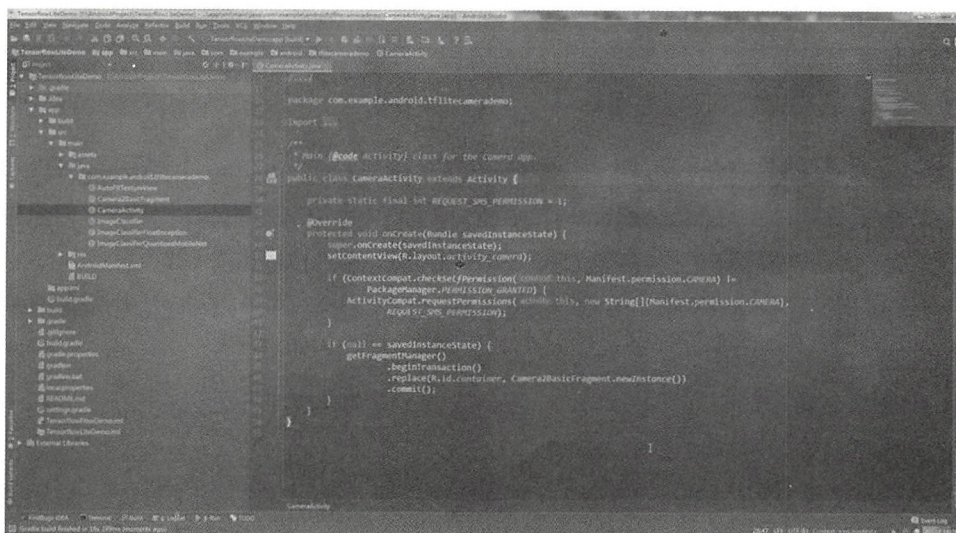


图 12-4

先来看一下这个项目的目录结构，如图 12-5 所示。

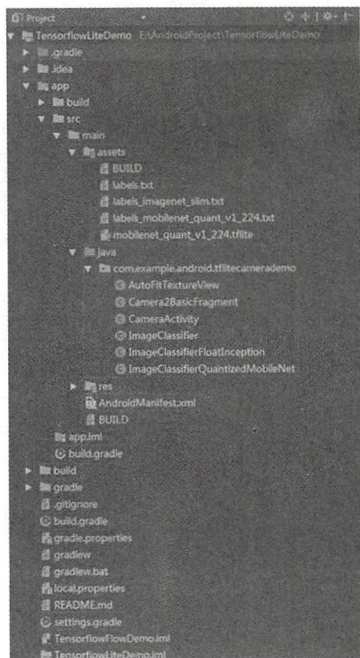


图 12-5

我们来简单地梳理下这个项目的目录结构，首先从大的目录来分，这个项目被划分为 5 个目录和若干文件。其中包括：

- **.gradle 目录和.idea 目录：**这两个目录是在项目导入 Android 工程后自动生成的。其中.gradle 目录中有 Android 项目运行或编译后产生的缓存文件；.idea 目录中有工作环境的配置文件，主要为 Android Studio 提供相关配置。
- **app 目录：**该目录为项目的主目录，与项目有关的所有代码、资源和文件等都会放在 app 目录下。
- **build 目录：**app 的构建目录，一般存放 app 每次构建（包括测试和正式发布）后所产生的 apk 文件。
- **gradle 目录：**Gradle 环境的支持文件目录，与.gradle 文件夹不同的是，这里的文件不是临时文件，而是固定的文件。

- **build.gradle** 文件: Gradle 项目自动编译时所需要的配置文件。
- **settings.gradle** 文件: Gradle 项目的子项目包含文件。

在开发时,编写的代码绝大多数都在 `app` 目录下。展开 `app` 目录后,可以看到里面还有若干个文件夹,接下来针对图 12-5 中有框的文件和文件夹进行讲解。可以看到,在 `src` 目录下又包含了一些子目录和文件,我们来说下这几个子目录和文件的作用。

- **main** 目录: 程序的主目录,其中的 `assets` 目录主要用来放置外部资源文件,如声音、视频、文本文件等,在这里放置 `tflite` 文件及其标签文件;
- **java** 目录: 代码文件的主目录,我们写的项目有关的代码一般都存放在 `java` 目录中;
- **res** 目录: 该目录同样也是资源目录,与 `assets` 目录不同的是, `res` 目录中所有的文件会自动生成一个唯一的 ID,而 `assets` 目录下的资源则不会;另外,一般将布局文件、图片文件、字符串配置文件等都放置在 `res` 目录下。`res` 目录下的配置文件大多以 `xml` 格式为主;
- **AndroidManifest.xml** 文件: 程序的主配置文件,一般整个 APP 的入口、图标、权限等都在这个文件中进行配置。

到目前为止,我们已经了解了 Android 的基本目录结构,以及每个目录和文件的作用,Android 的主要文件存放于 `app\main\java` 目录下。展开目录后可以看到项目的 `java` 目录下有一个子目录 `com.example.android.tflitecamerademo`,这个子目录实际上是一个包名,可以理解为 `com/example/android/tflitecamerademo`。实际上,所有的 `java` 文件都存放在 `tflitecamerademo` 目录下,展开这个目录,可以看到里面有 6 个子文件,它们分别是 `AutoFitTextureView`, `Camera2BasicFragment`, `CameraActivity`, `ImageClassifier`, `ImageClassifierFloatInception` 和 `ImageClassifierQuantizedMobileNet`,下面就来简单地介绍下这几个类。

AutoFitTextureView: 继承自 `TextureView` 的一个自定义控件,一个能用来自动适应特定样式比例的 `TextureView` 控件。`setAspectRatio()`是 `AutoFitTextureView` 控件里的设置视图长宽比的一个方法,视图将测量的大小根据比例计算参数,参数的实际

大小不重要，也就是说，调用设置长宽比 2:3 和设置长宽比 4:6 是同样的结果。其他的基本上都跟 TextureView 一样，在这个项目中显示的是相机的画面内容。¹

CameraActivity：用来展示页面的 Activity，里面只执行了一个操作，即加载了一个 Camera2BasicFragment，所以，直接看 Camera2BasicFragment 的解释就好。

Camera2BasicFragment：这是展示页面真正内容的类。一个 Fragment、页面加载，以及数据显示、数据处理、UI 界面上的内容全部都是在此类中进行的。在 onActivityCreated 中初始化 ImageClassifier 对象，也就是下面的重要的逻辑处理类，这个类主要用于加载模型并实现推理运算，然后打开一个后台线程 startBackgroundThread()，执行了死循环操作，重复执行 classifyFrame()这个方法，所以将目光聚焦在 classifyFrame()方法里。然后看到 ImageClassifier 的对象执行了一个 classifyFrame(bitmap, textToShow)的方法，并把数据存储在 SpannableStringBuilder 的对象 textToShow 里，调用 showToast(textToShow)方法。在 showToast(textToShow)方法里，textView.setText(builder, TextView.BufferType.SPANNABLE)将 builder 里面的数据显示在了页面上。

ImageClassifier：这是一个图像分类的抽象父类，大多数基本的图像分类操作都是在此类中进行的，在 Camera2BasicFragment 中获取图像数据，然后把数据传递到 ImageClassifier 中进行处理，处理完后将数据返回到 Camera2BasicFragment 中并显示在界面上。在 Camera2BasicFragment 里面调用 ImageClassifier 的 classifyFrame()方法，convertBitmapToByteBuffer(bitmap)就是关键的方法，将图像数据写进了 ByteBuffer，然后通过实现类实现的 runInference()方法去将图像数据存入 labelProbArray 中，最后 labelProbArray 转换为需要显示的文字。

1 TextureView：一个用来显示内容流的控件，这样的内容流可以是视频或者 OpenGL 的场景，也可以是本地应用及其他进程，但是它必须在硬件加速的窗口中运行，相对于展示视频和 OpenGL 的 SurfaceView 来说，能够进行 View 的一些变换操作，比如移动、设置动画等。

ImageClassifierFloatInception：这是 ImageClassifier 抽象类的实现类之一（本项目中没有用到）。这种图片分类器适用于 Inception-V3 的微型模型，它适用于浮点推理，而不适用于量化模型。

ImageClassifierQuantizedMobileNet：这是 ImageClassifier 抽象类的实现类之一（是本项目中用到的）。这种图片分类器适用于量子化的移动网络模型，适用于本项目。与上面的 Inception-V3 的图像分类模型类一样，它能够加载具体的 tflite 文件，以及移动网络量化的普通标签文件。

程序运行后效果如图 12-6 所示。当手机摄像头对准一个鼠标的时候，下面会显示所对准的物体是鼠标的概率为 98%。

到目前为止，我们已经在 Android 上实现了一个 TensorFlow 模型的加载调用。读者可以根据自己的数据集和相关需求对其进行更改，更符合自己的需求。

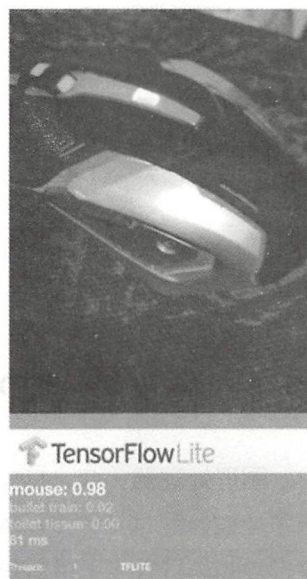


图 12-6

第 13 章

TensorFlow GPU

13.1 什么是 GPU

图形处理器（Graphics Processing Unit，GPU），又称显示核心、视觉处理器、显示芯片或绘图芯片，是一种专门在个人电脑、工作站、游戏机和移动设备（如平板电脑、智能手机等）上进行绘图运算工作的微处理器。在深度学习和人工智能领域，我们能够利用 GPU 加速计算功能来处理大量训练任务。而 GPU 加速计算是指同时利用图形处理器（GPU）和 CPU 来加快科学、分析、工程、消费和企业等应用程序的运行速度。GPU 加速器于 2007 年由 NVIDIA® 率先推出，现已在世界各地为政府实验室、高校、公司及中小型企业的高效能数据中心提供支持。GPU 能够使从汽车、手机和平板电脑，到无人机和机器人等平台的应用程序加速运行。

那么 GPU 与 CPU 的主要差别是什么呢？

首先，CPU 是由专为顺序串行处理而优化的几个核心组成的，这里包括我们比较熟悉的运算器（ALU）、寄存器组和状态寄存器；而 GPU 则拥有一个由数以千计的更小的、更高效的核心组成的大规模并行计算架构。GPU 采用了大量的计算单元和超长的流水线并拥有非常简单的控制逻辑，而对于 CPU 所拥有的缓存（Cache）却在 GPU 中几乎省略，在 GPU 中拥有很多个 ALU，而所剩下的极少 Cache 是为了给线程提供服务的，而非为了保存数据。

13.2 GPU 的选择

深度学习和人工智能是对计算要求非常高的应用领域, GPU 的选择将从根本上决定深度学习在研究过程中的体验甚至结果。在没有 GPU 的环境下, 如果想要训练一个较大的模型(例如训练聊天机器人 50 个 epoch), 可能需要 3 周左右的时间才能完成, 但是如果用 GPU 进行训练的话, 可能只需要不到 72 个小时。如果 GPU 的计算能力足够强大, 甚至使用 GPU 集群来进行数据训练的话, 也许只需要花费 24 个小时甚至更短的时间就可以完成训练。因此, 在深度学习和人工智能领域研究中选择一个合适的 GPU, 对日后的学习和训练起着事半功倍的作用。

GPU 的生产厂家有很多, 大家比较熟悉的有英特尔、英伟达、AMD 这三家。英特尔的 GPU 基本上都是集成显卡的芯片, 主要应用于英特尔的主板和 CPU 上, 所以, 英特尔的 GPU 无法作为深度学习的首选 GPU。

AMD 显卡来自于世界上第二大独立显卡的生产销售商, 其技术就是基于我们所熟知的 ATI 显卡, ATI 公司后被美国 AMD 公司收购, AMD 显卡也是很早之前的 A 卡。A 卡作为比较老牌的显卡, 在图像处理尤其是绘图方面有着不可替代的优越性。但是在做深度学习和人工智能领域的任务处理时, 由于 AMD 显卡没有相关的深度学习库, 所以 AMD 显卡在人工智能领域也无法作为首选显卡。

而英伟达(NVIDIA)开发出一个通用的并行计算架构——CUDA(Compute Unified Device Architecture), 该架构可以使 GPU 很容易地解决复杂的计算问题。在实际应用过程中一般使用 CUDA 中的 cuDNN 库。cuDNN 库是一个常见的神经网络加速库, 其作用就是把加载到显卡上的网络层数据进行优化计算。因此, 在深度学习和人工智能领域中, 常用 NVIDIA 来作为其主要训练显卡。

NVIDIA 显卡有很多, 从相对较低端的 GTX 750 到高端的 GTX 1080Ti, 再到更加高端的 Tesla 系列显卡。这些显卡在大家的眼里可能最大的区别就是价格相差非常大。但在深度学习的训练过程中, 显存却是非常重要的因素。而且, 影响深度学习训练结果的因素还有很多。接下来比较几个 NVIDIA 显卡的参数, 如图 13-1 所示。



参数信息

☒ 隐藏相同项

☒ 只显示选中项

品牌	MSI 微星	GALAXY (影驰)	Colorful (七彩虹)
品牌	微星 GeForce GTX 960 GAMING 2G	影驰 GeForce GTX 1060 GAMER 6G	七彩虹 iGame GTX 1080Ti Vulcan X OC
型号名称	微星 GeForce GTX 960 GAMING 2G	影驰 GeForce GTX 1060 GAMER 6G	七彩虹 iGame GTX 1080Ti Vulcan X OC
显卡核心			
芯片厂商	NVIDIA	NVIDIA	NVIDIA
显卡芯片	GeForce GTX 960	GeForce GTX 1060	GeForce GTX 1080Ti
显示芯片系列	NVIDIA GTX 900 系列	NVIDIA GTX 10 系列	NVIDIA GTX 10 系列
制作工艺	28 纳米	16 纳米	16 纳米
核心代号	GM206	GP106-400	GP102-350-K1-A1
核心频率	1216/1279 MHz	1556/1771 MHz	1480/1733 MHz
CUDA 核心	1024 个	1280 个	3584 个
显存规格			
显存频率	7010 MHz	8000 MHz	11000 MHz
显存类型	GDDR5	GDDR5	GDDR5X
显存容量	2 GB	6 GB	11 GB
显存位宽	128 bit	192 bit	352 bit
最大分辨率	4096 × 2160	7680 × 4320	

图 13-1

在图 13-1 中选用了 GTX 960、GTX 1060、GTX 1080Ti 三种显卡进行对比，从中可以发现，有一个参数是 CUDA 核心数。在深度学习和人工智能训练中，CUDA 核心数越高，代表着处理并发的性能越好。正如前面所说，GPU 和 CPU 的最大区别在于 GPU 中有着数以千计的计算单元，计算单元主要用于处理并行运算，而其计算单元的数量即为 CUDA 核心数；另外也可以发现，在显存规格中有一个叫显存位宽的参数，显存位宽大小与传输数据的速度和效率也是有很大关系的，一般来讲，显存位宽越大其性能越好；在深度学习训练过程中，显存容量也是一个非常关键的指标，显存容量的大小直接决定了在训练模型时可加载数据量的大小，因此，可以认为，在其他参数都相同的情况下，显存容量越大，对于深度学习的训练越有利。

13.3 搭建 TensorFlow GPU

在 13.2 节中我们了解了什么是 GPU，以及如何选择 GPU，现在就带领大家搭建 TensorFlow GPU。

要搭建 TensorFlow 的 GPU 版本，必备条件就是一块能够支持 CUDA 的 NVIDIA 显卡，因为在搭建 TensorFlow 的 GPU 版本时，首先需要安装其基础支持平台 CUDA 及其机器学习库 cuDNN，然后在此基础上搭建 TensorFlow GPU 版本。

其次还要了解一下不同的 TensorFlow 版本对应安装的 CUDA 和 cuDNN 版本，因为在 TensorFlow 的 GPU 版本安装过程中，如果对应的 CUDA 版本和 cuDNN 版本不正确的话，是无法正常使用 GPU 来进行模型训练的。表 13-1 整理了各个版本的 TensorFlow 所需要的 CUDA 和 cuDNN 的版本。

表 13-1

TensorFlow 版本	CUDA 版本	cuDNN 版本
1.2	CUDA Toolkit 8.0	cuDNN v5.1
1.3	CUDA Toolkit 8.0	cuDNN v6 or v6.1
1.4	CUDA Toolkit 8.0	cuDNN v6.1
1.5	CUDA Toolkit 9.0	cuDNN v7.0
1.6	CUDA Toolkit 9.0	cuDNN v7.0
1.7	CUDA Toolkit 9.0	cuDNN v7.0
1.8	CUDA Toolkit 9.0	cuDNN v7.0

接下来就从 Windows 和 Linux 两个平台出发，分别带领大家来搭建 TensorFlow GPU 版本。

13.3.1 在 Windows 上搭建 TensorFlow GPU

前面已经讲解了如何安装 Python 环境，在此默认大家已经安装好了 Anaconda 和 Python 3.6 版本，直接从 CUDA 安装开始讲解。

先来说一下整体搭建所需要的环境，在这里以 TensorFlow 1.7 版本为例，所需要的环境如下。

- Anaconda 3(64bit)
- Windows 10(64bit)
- CUDA-9.0
- cuDNN-7.0
- Python-3.6
- TensorFlow-gpu 1.7
- NVIDIA GeForce GTX 1080Ti (这是作者的显卡型号)

前提准备条件（所有内容全为 64 位）

(1) Python-3.6。

(2) Anaconda.3-5.1.0 (64 位)。

(3) 显卡驱动：建议去英伟达官方网站下载，驱动不要最新的，时间最好跟 CUDA 和 cuDNN 发布时间相差无几。

(4) CUDA-9.0.176_win10 (V9.0)。

此外包括了 CUDA9.0.176 的两个补丁 CUDA-9.0.176.1_Windows 和 CUDA-9.0.176.2_Windows。

(5) cuDNN-9.0-Windows10-x64-v7 (其实从官网下载的是 V7.0，需要注册账号)。

安装过程

首先，使用 CUDA 官方网站所下载的 Windows 版本的安装包安装 CUDA 的主程序包：CUDA-9.0.176_win10。在官方网站下载时，还有两个补丁包，需要一起下载下来，如图 13-2 所示。

03.cuda_9.0.176_win10	2018/5/18 17:25	应用程序	1,398,676 KB
04.cuda_9.0.176.1_windows	2018/5/18 17:23	应用程序	54,094 KB
05.cuda_9.0.176.2_windows	2018/5/18 17:23	应用程序	54,673 KB

图 13-2

在 CUDA-9.0.176_win10 上单击鼠标右键，以管理员身份运行。运行后，在进行完系统检查后会自动跳转到许可协议界面。在许可协议界面中，单击“同意并继续”按钮进行下一步安装，如图 13-3 所示。

如图 13-4 所示，在选择安装选项的时候，建议使用自定义选项，然后单击“下一步”按钮，如图 13-5 所示。

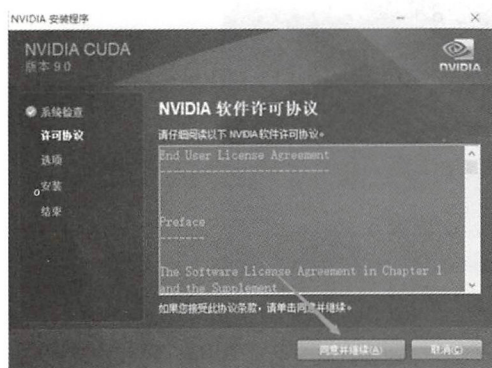


图 13-3

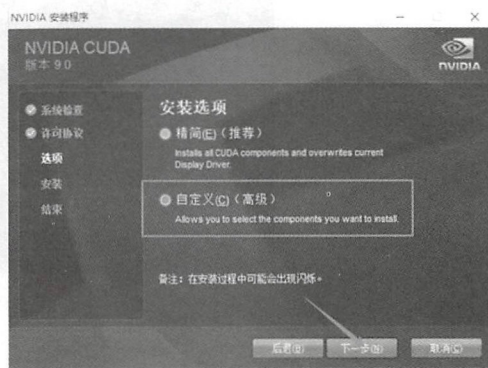


图 13-4

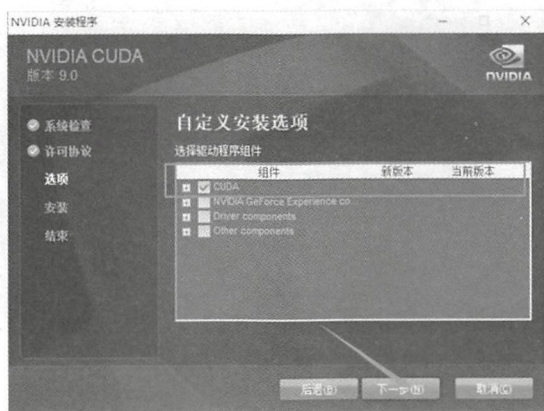


图 13-5

由于电脑一般在安装完系统之后就会将显卡的驱动安装上，所以在这里，只需要安装 CUDA 组件。如果安装第二项显卡驱动，则很有可能和当前显卡驱动冲突，导致最后安装失败，或安装后无法使用、黑屏等现象。选择完 CUDA 之后，单击“下一步”按钮选择安装位置后，继续单击“下一步”按钮即可开始安装，分别如图 13-6、图 13-7、图 13-8 所示。

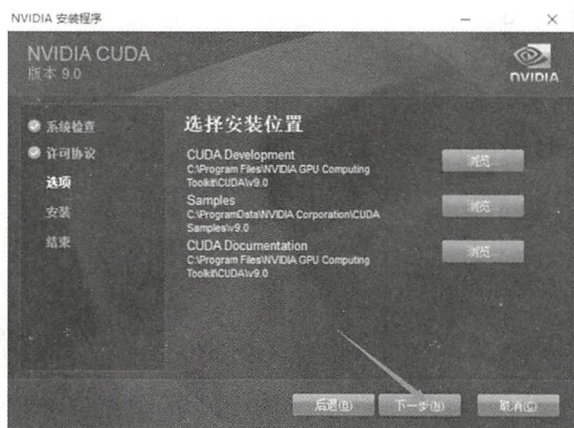


图 13-6

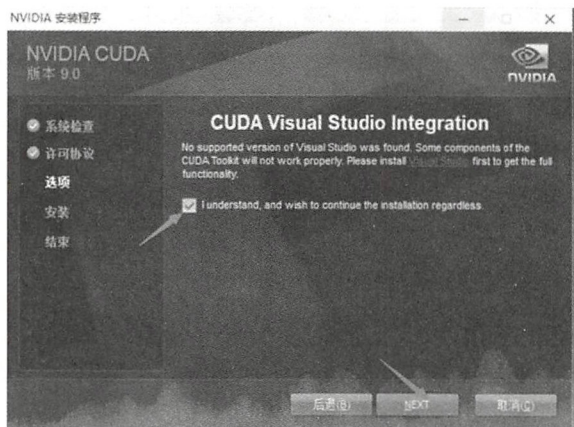


图 13-7

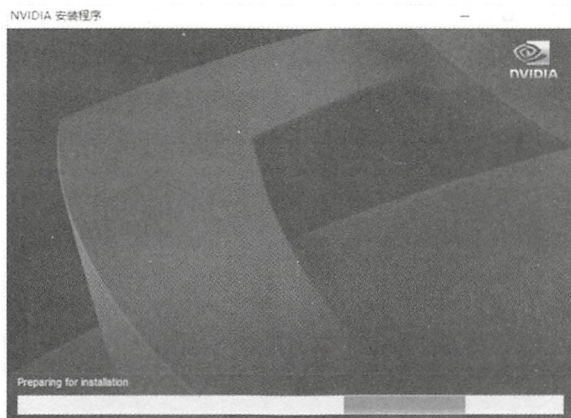


图 13-8

经过一段时间的安装后，就会提示安装完成，此时单击“下一步”按钮后显示“已安装程序”，单击“关闭”按钮，安装完成，如图 13-9 所示。

此时，CUDA 的安装部分就已经完成。接下来需要安装两个补丁包，安装补丁包的方法和安装 CUDA 的方法大同小异，在此就不做过多讲解，安装完成后，如图 13-10 所示（两个补丁包安装成功后的界面是一样的）。

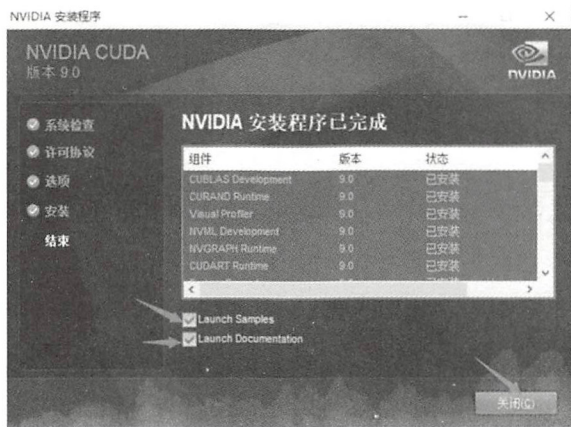


图 13-9



图 13-10

安装完 CUDA 之后，接下来要做的就是安装 cuDNN 程序。安装 cuDNN 程序相对比较简单，只需要在 cuDNN 的官方网站上下载对应的 zip 版本压缩包，解压后拷贝里面的 bin, include, lib 文件夹和一个 txt 文件到 CUDA9.0 的根目录下即可（如果按照上面默认安装的话，路径是 C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v9.0），如图 13-11 所示。



图 13-11

当 CUDA 和 cuDNN 安装完成后，接下来的步骤就是配置环境变量。配置环境变量也是在整个安装过程中相对比较核心的步骤，很多读者在实际过程中安装好了 CUDA 和 cuDNN 后发现无法使用 TensorFlow 的 GPU 版本基本上也都是因为没有配

置环境变量。配置环境变量也比较简单：

```
CUDA_SDK_PATH = C:\ProgramData\NVIDIA Corporation\CUDA Samples\v9.0
CUDA_LIB_PATH = %CUDA_PATH%\lib\x64
CUDA_BIN_PATH = %CUDA_PATH%\bin
CUDA_SDK_BIN_PATH = %CUDA_SDK_PATH%\bin\win64
CUDA_SDK_LIB_PATH = %CUDA_SDK_PATH%\common\lib\x64
```

配置完成后，在 path 环境变量中增加如下参数：

```
%CUDA_LIB_PATH%
%CUDA_BIN_PATH%
%CUDA_SDK_LIB_PATH%
%CUDA_SDK_BIN_PATH%
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v9.0\lib\x64;
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v9.0\bin;
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v9.0\common\lib\x64;
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v9.0\bin\win64;
```

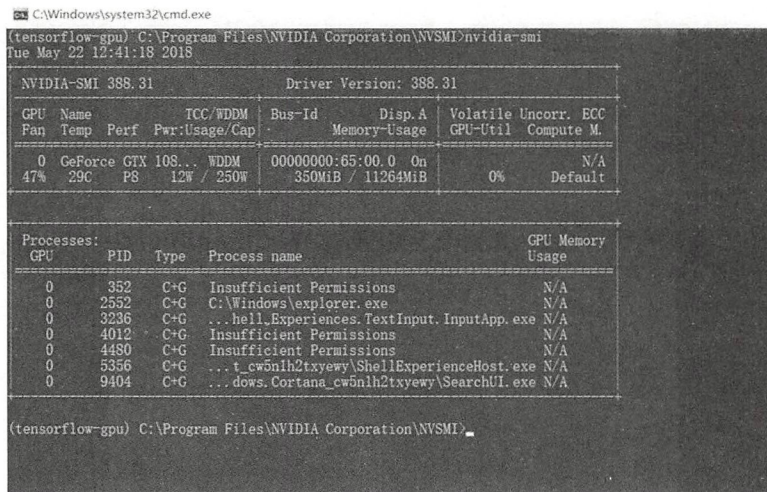
增加后如图 13-12 所示。



图 13-12

在这里要注意的是 C:\ProgramData\NVIDIA Corporation\CUDA Samples\v9.0\是作者的安装目录，如果在安装过程中将目录设置为其他目录的话，则需要做相应的改变。

至此，环境变量也配置完成，可测试下所搭建的环境是否正确。首先使用 `cmd` 命令进入 `C:\ProgramData\NVIDIA Corporation\NVSMI\` 目录，然后输入 `nvidia-smi`，如果出现如图 13-13 所示的信息，则说明安装配置正确。



```
C:\Windows\system32\cmd.exe
(tensorflow-gpu) C:\Program Files\NVIDIA Corporation\NVSMI>nvidia-smi
Tue May 22 12:41:18 2018
```

NVIDIA-SMI 388.31				Driver Version: 388.31			
GPU	Name	TCG/WDMM	Bus-Id	Disp.A	Memory-Usage	Volatile Uncorr. ECC	GPU-Util
Fan	Temp	Perf	Pwr:Usage/Cap			GPU-Util	Compute M.
0	GeForce GTX 1080	WDDM	00000000:65:00:0	On	350MiB / 11264MiB	0%	N/A
47%	29C	P8	12W / 250W				Default

Processes:					GPU Memory Usage
GPU	PID	Type	Process name		
0	352	C+G	Insufficient Permissions		N/A
0	2552	C+G	C:\Windows\explorer.exe		N/A
0	3236	C+G	...hell.Experiences.TextInput.InputApp.exe		N/A
0	4012	C+G	Insufficient Permissions		N/A
0	4480	C+G	Insufficient Permissions		N/A
0	5356	C+G	...t_cw5nlh2txyewy\ShellExperienceHost.exe		N/A
0	9404	C+G	...dows.Cortana_cw5nlh2txyewy\SearchUI.exe		N/A

```
(tensorflow-gpu) C:\Program Files\NVIDIA Corporation\NVSMI>
```

图 13-13

接下来就要进行 TensorFlow 的 GPU 版本安装了，使用 Anaconda 安装 GPU 版本的过程和安装 CPU 版本的过程一样，使用 `pip` 安装时只需简单修改一下命令即可。

```
pip install tensorflow-gpu==1.7.0
```

安装完成后，就可以使用 Anaconda 来测试下 GPU 是否安装成功。我们可以随意找一段 TensorFlow 的代码运行，如果正常，且提示是使用 GPU 运行的则代表安装成功。使用 GPU 方式运行会有明显的标志，如图 13-14 所示。

从图 13-14 可以看到，使用 GPU 方式运行时会明确标出具体使用的 GPU 型号、剩余内存使用量等。

至此，我们的 Windows 环境下搭建的 TensorFlow GPU 版本已经完成了。

```
C:\Windows\system32\cmd.exe - python
>>>
>>>
>>>
>>> import tensorflow as tf
>>> a = tf.constant(3)
>>> b = tf.constant(2)
>>> print(a, b)
Tensor("Const:0", shape=(), dtype=int32) Tensor("Const_1:0", shape=(), dtype=int32)
>>> c = a + b
>>> print(c)
Tensor("add:0", shape=(), dtype=int32)
>>> sess = tf.Session()
2018-05-22 16:14:05.844688: I T:\src\github\tensorflow\tensorflow\core\platform\cpu_feature_guard.cc:140] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2
2018-05-22 16:14:06.117794: I T:\src\github\tensorflow\tensorflow\core\common_runtime\gpu\gpu_device.cc:1344] Found device 0 with properties:
  name: GeForce GTX 1080 Ti major: 6 minor: 1 memoryClockRate(GHz): 1.645
  pciBusID: 0000:65:00:0
  totalMemory: 11.00GiB freeMemory: 9.10GiB
2018-05-22 16:14:06.121077: I T:\src\github\tensorflow\tensorflow\core\common_runtime\gpu\gpu_device.cc:1423] Adding visible gpu devices: 0
2018-05-22 16:14:06.650412: I T:\src\github\tensorflow\tensorflow\core\common_runtime\gpu\gpu_device.cc:911] Device interconnect StreamExecutor with strength 1 edge matrix:
2018-05-22 16:14:06.652524: I T:\src\github\tensorflow\tensorflow\core\common_runtime\gpu\gpu_device.cc:917]      0
2018-05-22 16:14:06.654417: I T:\src\github\tensorflow\tensorflow\core\common_runtime\gpu\gpu_device.cc:930] 0:  N
2018-05-22 16:14:06.656259: I T:\src\github\tensorflow\tensorflow\core\common_runtime\gpu\gpu_device.cc:1041] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 8806 MB memory) -> physical GPU (device: 0, name: GeForce GTX 1080 Ti, pci bus id: 0000:65:00:0, compute capability: 6.1)
>>> sess.run((a,b))
(3, 2)
>>> print(sess.run(c))
5
>>>
```

图 13-14

13.3.2 在 Linux 上搭建 TensorFlow GPU

在 Linux 下搭建 TensorFlow GPU 环境和在 Windows 下搭建 TensorFlow GPU 环境实际上是大同小异的，其基本步骤大体相同，都是先安装显卡，然后安装 CUDA、cuDNN，接下来再搭建 Python 环境并安装 TensorFlow 的 GPU 版本。

仍然先说一下整体搭建所需要的环境，在这里以 TensorFlow 1.7 版本为例，所需要的环境如下：

- Ubuntu 16.04(64bit)
- CUDA-9.0
- cuDNN-7.0
- Python-3.6
- TensorFlow-gpu 1.6
- NVIDIA P5000（这是作者显卡的型号）

在安装环境之前，首先使用 `lspci | grep -i nvidia` 来检查一下系统中是否包含 NVIDIA 显卡，如图 13-15 所示。

```
03:00.0 VGA compatible controller: NVIDIA Corporation Device 1bb0 (rev a1)
03:00.1 Audio device: NVIDIA Corporation Device 10f0 (rev a1)
```

图 13-15

如果系统中有可使用的显卡则会显示，否则会为空白，如图 13-16 所示。

```
# lspci | grep -i nvidia
#
```

图 13-16

在确认系统中有可使用的显卡后，接下来就要进行显卡驱动安装。一般来讲，在安装 CUDA 的时候会自带显卡驱动的安装步骤，但是由于 CUDA 中所带的显卡驱动可能会有一定的不兼容问题，因此，一般建议手动安装显卡驱动。如果不清楚要安装什么版本的驱动，可以使用 `ubuntu-drivers devices` 命令确认，如图 13-17 所示。

```
== /sys/devices/pci0000:00/0000:00:02.0/0000:03:00.0 ==
vendor : NVIDIA Corporation
modalias : pci:v000010DEd00001B80sv00001028sd000011B2bc03sc00i00
driver : xserver-xorg-video-nouveau - distro free builtin
driver : nvidia-396 - third-party non-free recommended
driver : nvidia-387 - third-party non-free
driver : nvidia-390 - third-party non-free
driver : nvidia-384 - distro non-free
```

图 13-17

图 13-17 使用的是 384 版本的显卡，因此，我们可以在命令行中输入 `sudo apt-get install nvidia-384` 命令进行显卡安装，安装后需要重启系统以使显卡生效。重启后，在命令行输入 `nvidia-smi`，如果出现如图 13-18 所示的界面，则说明安装成功。

```
NVIDIA-SMI 384.90 Driver Version: 384.90
+-----+
| GPU   | Name      | Persistence-M | Bus-Id  | Disp.A | Volatile Uncorr. ECC |
| Fan   | Temp  Perf | Pwr:Usage/Cap |          | Memory-Usage | GPU-Util  Compute M. |
+-----+
| 0     | Quadro P5000 | Off          | 00000000:03:00.0 Off | 0MB / 16273MB | 1%      Default |
+-----+

Processes:
+-----+
| GPU   | PID  Type  Process name      | GPU Memory Usage |
+-----+
| No running processes found |
+-----+
```

图 13-18

安装完显卡后，接下来就到了非常重要的环节，那就是安装 CUDA 和 cuDNN。可以在 <https://developer.nvidia.com/cuda-90-download-archive> 网站上下载 CUDA 的 9.0 版本，这里值得注意的是，千万不要直接进官网下载最新版本的 CUDA，目前最新版本是 9.2，安装后会导致 TensorFlow GPU 版本无法使用，如图 13-19 所示。

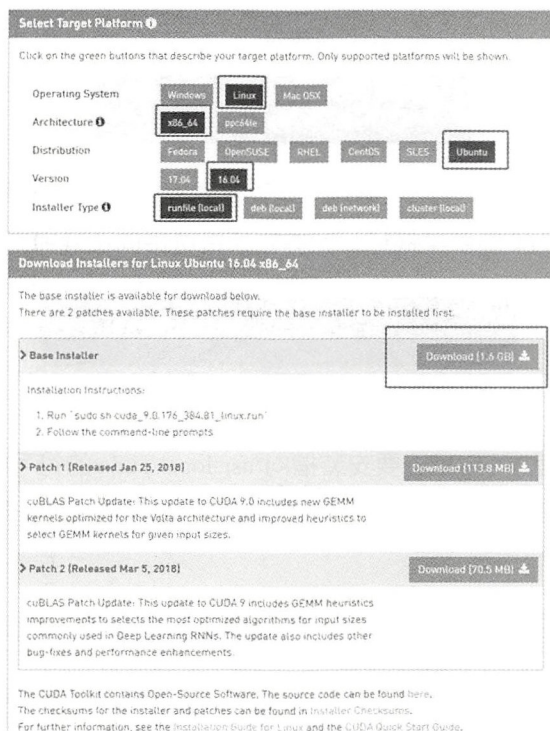


图 13-19

在这里，依次选择 Linux→x86_64→Ubuntu→16.04→runfile(local)后，下载 Base Installer 版本，下载后使用 `sudo sh ./cuda_9.0.176_384.81_linux.run` 命令进行安装。在安装开始首先会有一大堆的条款，在这里一直按住回车键，到 100%后，会问你是否接受条款，此时输入 `accept`，表示接受，如图 13-20 所示。

```
Do you accept the previously read EULA?
accept/decline/quit: accept
```

图 13-20

接下来会问你是否需要安装显卡，在这里一定要选择 n，如图 13-21 所示。

```
Install NVIDIA Accelerated Graphics Driver for Linux-x86_64 384.81?  
(y)es/(n)o/(q)uit: n
```

图 13-21

接下来会询问是否需要安装 CUDA 工具，这里选择 y，如图 13-22 所示。

```
Install the CUDA 9.0 Toolkit?  
(y)es/(n)o/(q)uit: y
```

图 13-22

接下来输入 CUDA 的安装地址，如果默认直接按回车键即可，如图 13-23 所示。

```
Enter Toolkit Location  
[ default is /usr/local/cuda-9.0 ]:
```

图 13-23

接下来还会继续询问你是否要安装指向/usr/local/cuda 的符号链接，以及是否安装 samples 等信息。这些都默认使用“y”同意安装，在选择路径时，直接按回车键即可，等待安装结束。

安装结束后，在/etc/profile 文件下加入相关的环境变量，如图 13-24 所示。

至此，CUDA 就安装成功了，安装成功后，接下来要安装 cuDNN。cuDNN 的安装方法非常简单，首先进入 cuDNN 的官网，<https://developer.nvidia.com/rdp/cudnn-archive> 并注册，找到 7.0 版本进行下载即可，如图 13-25 所示。

```
export LD_LIBRARY_PATH=/usr/local/cuda-9.0/lib64:/usr/local/cuda/extras/CUPTI/lib64  
export CUDA_HOME=/usr/local/cuda-9.0  
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64  
export CLASSPATH=.:$JAVA_HOME/jre/lib/rt.jar:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar  
export PATH=/usr/local/cuda-9.0/bin:$JAVA_HOME/bin:$PATH
```

图 13-24

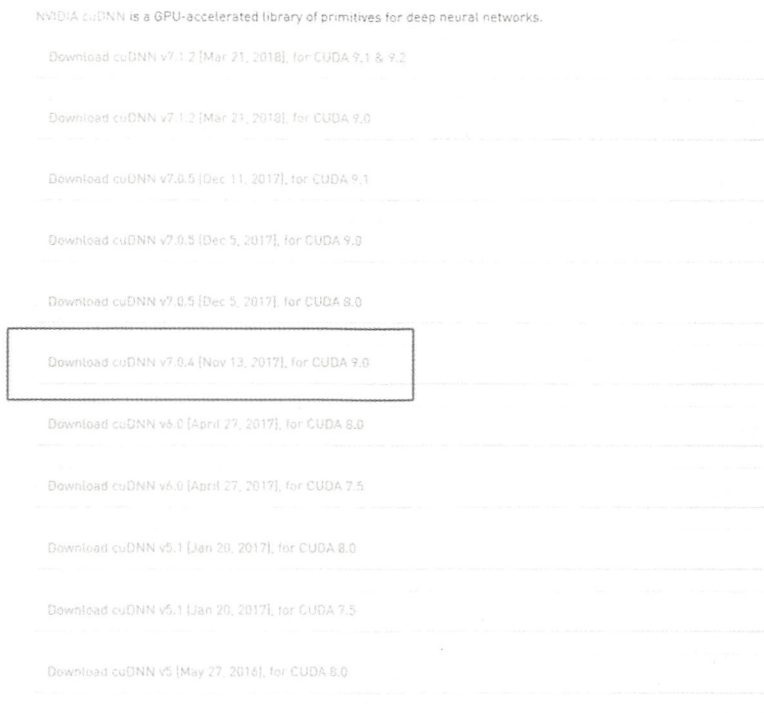


图 13-25

下载完成后，使用 `tar -zxvf` 命令进行解压，并将解压后的文件复制到 CUDA 对应的安装目录即可完成安装。

至此，显卡+CUDA+cuDNN 的所有安装都已经完成，接下来就可以使用 `pip install tensorflow-gpu==1.6` 命令安装即可。安装完成后，进入 Python 环境，导入 TensorFlow 包，并输出最后的版本，如果输出成功，则代表安装成功。

```
>>> import tensorflow as tf
>>> print(tf.__version__)
1.6.0
```

13.4 使用 TensorFlow GPU 进行训练

在使用 TensorFlow GPU 进行训练的时候，可以指定哪一部分的训练使用哪一个

GPU 或者 CPU。如果没有指定 GPU, TensorFlow 则会默认使用第一块 GPU 进行训练。

在 TensorFlow 中, 我们可以使用 `tf.device()` 函数来指定使用什么设备进行训练。例如, 要指定使用电脑的第一块 GPU 进行训练, 就可以设定 `tf.device('/gpu:0')`, 同样地, 如果我们要使用第二块 GPU 进行训练 (前提是你的设备中有两块 GPU), 那么可以设定 `with tf.device('/gpu:1')`。如果要使用 CPU 进行训练, 则可以设定 `tf.device('/cpu:0')`。

下面来看一个对比例子, 首先不指定 GPU 进行一个简单的计算。代码如下:

```
import tensorflow as tf
A = tf.Variable([[1, 2], [3, 4]], dtype = tf.float32, name='A')
B = tf.Variable([[1, 1], [1, 1]], dtype = tf.float32, name='B')
y = tf.matmul(A, B)
z = tf.sigmoid(y)
print(z)

init_op = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init_op)
    z = sess.run(z)
    print(z)
```

运行效果如图 13-26 所示。

```
Tensor("Sigmoid:0", shape=(2, 2), dtype=float32)
2018-05-22 17:49:58.719434: I tensorflow/core/platform/cpu_feature_guard.cc:140] Your CPU supports instructions that
2018-05-22 17:49:59.603582: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1212] Found device 0 with properties:
name: Quadro P5000 major: 6 minor: 1 memoryClockRate(GHz): 1.7335
pciBusID: 0000:03:00.0
totalMemory: 15.89GiB freeMemory: 15.77GiB
2018-05-22 17:49:59.603617: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1312] Adding visible gpu devices: 0
2018-05-22 17:49:59.838181: I tensorflow/core/common_runtime/gpu/gpu_device.cc:993] Creating TensorFlow device (/job
y: 6.1)
[[ 0.95257413  0.95257413]
 [ 0.999089    0.999089   ]]
```

图 13-26

第一行是打印出 `z` 的值。接下来会提示找到了一个设备 0, 名字为 Quadro P5000, 这是 GPU 的型号。然后使用 GPU 开始训练, 接下来把代码稍微改一改。

```
import tensorflow as tf
with tf.device('/cpu:0'):
    A = tf.Variable([[1, 2], [3, 4]], dtype = tf.float32, name='A')
    B = tf.Variable([[1, 1], [1, 1]], dtype = tf.float32, name='B')
    y = tf.matmul(A, B)
```

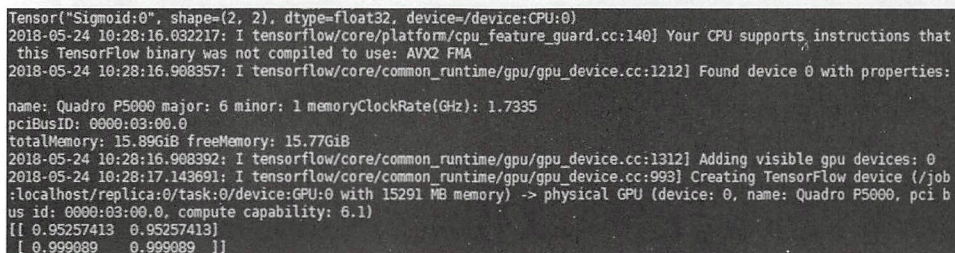
```

z = tf.sigmoid(y)
print(z)

init_op = tf.global_variables_initializer()
with tf.Session() as sess:
    with tf.device('/gpu:0'):
        sess.run(init_op)
        z = sess.run(z)
        print(z)

```

在这段代码中，我们把定义部分和计算部分放在了 CPU 中，然后把后面的 sess.run() 运行部分放在了 GPU 中，运行下这段代码，看看会发生什么变化，如图 13-27 所示。



```

Tensor("Sigmoid:0", shape=(2, 2), dtype=float32, device=/device:CPU:0)
2018-05-24 10:28:16.032217: I tensorflow/core/platform/cpu_feature_guard.cc:140] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2 FMA
2018-05-24 10:28:16.908357: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1212] Found device 0 with properties:
name: Quadro P5000 major: 6 minor: 1 memoryClockRate(GHz): 1.7335
pciBusID: 0000:03:00.0
totalMemory: 15.89GiB freeMemory: 15.77GiB
2018-05-24 10:28:16.908392: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1312] Adding visible gpu devices: 0
2018-05-24 10:28:17.143691: I tensorflow/core/common_runtime/gpu/gpu_device.cc:993] Creating TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 15291 MB memory) -> physical GPU (device: 0, name: Quadro P5000, pci b
us id: 0000:03:00.0, compute capability: 6.1)
[[ 0.95257413  0.95257413]
 [ 0.999089   0.999089  ]]

```

图 13-27

第一行同样是将 z 的值打印出来，不过对比图 13-26 发现，所输出的结果后面多了一句 device=/device:CPU:0，通过这句话可以知道，“with tf.device('/cpu:0')”起作用了。也就是说，A、B、y、z 这几个参数我们是放在了 CPU 下进行定义并运算的，其运算结果 z 也是使用 CPU 进行处理的，然后接下来在运行部分设置使用了 GPU，因此，最后打印出来的结果是使用 GPU 进行运算的，由此可见，tf.device() 函数可以很方便地设定具体使用 CPU 还是 GPU。

值得注意的是，在 TensorFlow 中，不能将 GPU 不支持的运算部分强行地放到 GPU 中运算，否则会报错：

```

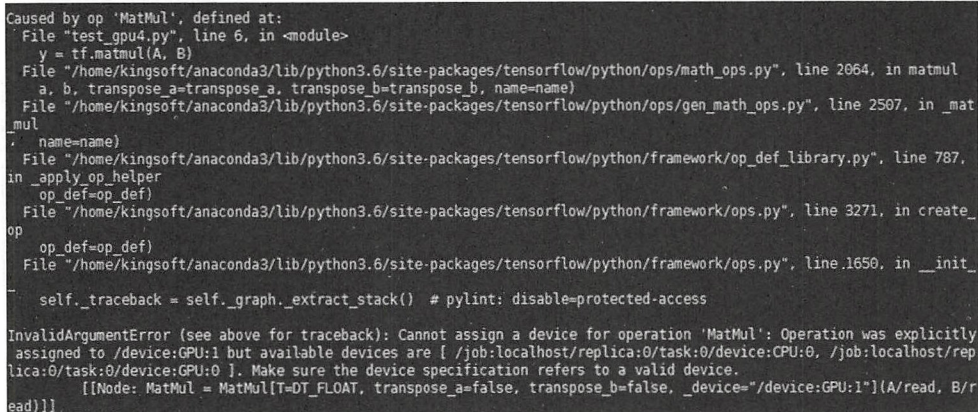
import tensorflow as tf
with tf.device('/cpu:0'):
    A = tf.Variable([[1, 2], [3, 4]], dtype = tf.float32, name='A')
    B = tf.Variable([[1, 1], [1, 1]], dtype = tf.float32, name='B')
with tf.device('/gpu:1'):

```



```
y = tf.matmul(A, B)
z = tf.sigmoid(y)
init_op = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init_op)
    z = sess.run(y)
    print(y)
```

将运算的部分放到了 GPU 中，会报如图 13-28 所示的错误。从图 13-28 中可以看到，如果 MatMul（不能在 GPU 上运行）运算部分被强行地放到 GPU 中执行，程序就会报错，并自动退出。



```
Caused by op 'MatMul', defined at:
  File "test_gpu4.py", line 6, in <module>
    y = tf.matmul(A, B)
  File "/home/kingsoft/anaconda3/lib/python3.6/site-packages/tensorflow/python/ops/math_ops.py", line 2064, in matmul
    a, b, transpose_a=transpose_a, transpose_b=transpose_b, name=name)
  File "/home/kingsoft/anaconda3/lib/python3.6/site-packages/tensorflow/python/ops/gen_math_ops.py", line 2507, in _matmul
    name=name)
  File "/home/kingsoft/anaconda3/lib/python3.6/site-packages/tensorflow/python/framework/op_def_library.py", line 787,
in _apply_op_helper
    op_def=op_def)
  File "/home/kingsoft/anaconda3/lib/python3.6/site-packages/tensorflow/python/framework/ops.py", line 3271, in create_
op
    op_def=op_def)
  File "/home/kingsoft/anaconda3/lib/python3.6/site-packages/tensorflow/python/framework/ops.py", line 1650, in __init_
    self.traceback = self.graph.extract_stack() # pylint: disable=protected-access
InvalidArgumentError (see above for traceback): Cannot assign a device for operation 'MatMul': Operation was explicitly
assigned to /device:GPU:1 but available devices are [ /job:localhost/replica:0/task:0/device:CPU:0, /job:localhost/rep
lica:0/task:0/device:GPU:0 ]. Make sure the device specification refers to a valid device.
[[Node: MatMul = MatMul[T=DT_FLOAT, transpose_a=false, transpose_b=false, _device="/device:GPU:1"]](A/read, B/r
ead)]]
```

图 13-28

一般来讲，GPU 支持操作 float32 等浮点型，但是如果要把一些浮点型和其他操作混合使用，则可以用另外一种在 GPU 中执行运算的方法，即设置 config=tf.ConfigProto(allow_soft_placement=True, log_device_placement=True)。这个方法实际上是一种兼容性的操作，目的是通过 allow_soft_placement 参数将无法放在 GPU 上的操作自动放回 CPU 上，这样就可以使能在 GPU 中操作的部分在 GPU 中运算，其他的运算会自动放回到 CPU 中运算，代码如下：

```
import tensorflow as tf
with tf.device('/cpu:0'):
    A = tf.Variable([[1, 2], [3, 4]], dtype = tf.float32, name='A')
    B = tf.Variable([[1, 1], [1, 1]], dtype = tf.float32, name='B')
with tf.device('/gpu:1'):
```

```

y = tf.matmul(A, B)
z = tf.sigmoid(y)
init_op = tf.global_variables_initializer()
with tf.Session() as sess:
    sess = tf.Session(config=tf.ConfigProto(
        allow_soft_placement=True, log_device_placement = True))
    sess.run(init_op)
    z = sess.run(y)
    print(y)

```

在上述代码中，在 GPU 中进行了两个操作：第一个操作是做了一个 MatMul 运算；第二个操作是进行了一个 Sigmoid 运算，并且也设置了 `config=tf.ConfigProto(allow_soft_placement=True, log_device_placement = True)` 命令，此时运行代码，得到如图 13-29 所示的结果。

```

2018-05-24 11:39:55.741776: I tensorflow/core/platform/cpu_feature_guard.cc:140] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2 FMA
2018-05-24 11:39:56.632633: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1212] Found device 0 with properties:
name: Quadro P5000 major: 6 minor: 1 memoryClockRate(GHz): 1.7335
pciBusID: 0000:03:00:0
totalMemory: 15.89GiB freeMemory: 15.77GiB
2018-05-24 11:39:56.632665: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1312] Adding visible gpu devices: 0
2018-05-24 11:39:56.860876: I tensorflow/core/common_runtime/gpu/gpu_device.cc:993] Creating TensorFlow device (/job:ldc
alhost/replica:0/task:0/device:GPU:0 with 15291 MB memory) -> physical GPU (device: 0, name: Quadro P5000, pci bus id: 0
000:03:00:0, compute capability: 6.1)
2018-05-24 11:39:56.939823: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1312] Adding visible gpu devices: 0
2018-05-24 11:39:56.939900: I tensorflow/core/common_runtime/gpu/gpu_device.cc:993] Creating TensorFlow device (/job:loc
alhost/replica:0/task:0/device:GPU:0 with 579 MB memory) -> physical GPU (device: 0, name: Quadro P5000, pci bus id: 000
0:03:00:0, compute capability: 6.1)
Device mapping:
/job:localhost/replica:0/task:0/device:GPU:0 -> device: 0, name: Quadro P5000, pci bus id: 0000:03:00:0, compute capabil
ity: 6.1
2018-05-24 11:39:56.939116: I tensorflow/core/common_runtime/direct_session.cc:297] Device mapping:
/job:localhost/replica:0/task:0/device:GPU:0 -> device: 0, name: Quadro P5000, pci bus id: 0000:03:00:0, compute capabil
ity: 6.1
B: (VariableV2): /job:localhost/replica:0/task:0/device:CPU:0
2018-05-24 11:39:56.939870: I tensorflow/core/common_runtime/placer.cc:875] B: (VariableV2)/job:localhost/replica:0/task
:0/device:CPU:0
B/read: (Identity): /job:localhost/replica:0/task:0/device:CPU:0
2018-05-24 11:39:56.939884: I tensorflow/core/common_runtime/placer.cc:875] B/read: (Identity)/job:localhost/replica:0/t
ask:0/device:CPU:0
B/Assign: (Assign): /job:localhost/replica:0/task:0/device:CPU:0
2018-05-24 11:39:56.939895: I tensorflow/core/common_runtime/placer.cc:875] B/Assign: (Assign)/job:localhost/replica:0/t
ask:0/device:CPU:0
A: (VariableV2): /job:localhost/replica:0/task:0/device:CPU:0
2018-05-24 11:39:56.939904: I tensorflow/core/common_runtime/placer.cc:875] A: (VariableV2)/job:localhost/replica:0/task
:0/device:CPU:0
A/read: (Identity): /job:localhost/replica:0/task:0/device:CPU:0
2018-05-24 11:39:56.939912: I tensorflow/core/common_runtime/placer.cc:875] A/read: (Identity)/job:localhost/replica:0/t
ask:0/device:CPU:0
MatMul: (MatMul): /job:localhost/replica:0/task:0/device:GPU:0
2018-05-24 11:39:56.939924: I tensorflow/core/common_runtime/placer.cc:875] MatMul: (MatMul)/job:localhost/replica:0/tas
k:0/device:GPU:0
A/Assign: (Assign): /job:localhost/replica:0/task:0/device:GPU:0
2018-05-24 11:39:56.939932: I tensorflow/core/common_runtime/placer.cc:875] A/Assign: (Assign)/job:localhost/replica:0/t
ask:0/device:CPU:0
init: (NoOp): /job:localhost/replica:0/task:0/device:CPU:0
2018-05-24 11:39:56.939939: I tensorflow/core/common_runtime/placer.cc:875] init: (NoOp)/job:localhost/replica:0/task:0/
device:CPU:0
B/initial_value: (Const): /job:localhost/replica:0/task:0/device:CPU:0
2018-05-24 11:39:56.939948: I tensorflow/core/common_runtime/placer.cc:875] B/initial_value: (Const)/job:localhost/repli
ca:0/task:0/device:CPU:0
A/initial_value: (Const): /job:localhost/replica:0/task:0/device:CPU:0
2018-05-24 11:39:56.939955: I tensorflow/core/common_runtime/placer.cc:875] A/initial_value: (Const)/job:localhost/repli
ca:0/task:0/device:CPU:0
[[ 3.  3.]
 [ 7.  7.]]

```

图 13-29

从图 13-29 中很容易看出哪个操作被放到了哪个设备上执行，并且最终得到了执行的结果。

这是在 TensorFlow 训练时进行 GPU 调用中最常使用的方式。

第 14 章

TensorFlow 与目标检测

我们经常会在很多工业领域和民用领域中碰到一些需要对图像中的物体进行识别或检测的需求，例如，从一张有上千人的照片中数出人数，甚至数出其中有多少男性、多少女性；再比如，从一张动物照片中识别出一共有多少种动物，并且每种动物有多少只。以上的种种需求，在计算机视觉领域中有一个统一的名词——目标检测。

目标检测，也叫目标提取，是一种基于目标几何和统计特征的图像分割。它将目标的分割和识别合二为一，其准确性和实时性是整个系统的重要能力。尤其在复杂场景中需要对多个目标进行实时处理时，目标自动提取和识别就显得特别重要。随着计算机技术的发展和计算机视觉原理的广泛应用，计算机图像处理技术对目标进行实时跟踪定位的研究越来越热门，在智能化交通系统、智能监控系统、军事目标检测及医学导航手术器械定位等方面具有广泛的应用价值。

14.1 传统目标检测方法

在深度学习和机器学习兴起之前，人们利用机器视觉进行目标检测的方法相对比较传统，大部分目标检测系统都是基于滑动窗口结合分类算法实现的。

传统的目标检测方法大多数都是基于统计学的知识而来的，所检测的对象相对来

讲也比较局限，以人脸检测和车牌检测为主。当给定一个图像，需要检测图像中的某个物体时，传统的目标检测方法首先会确定一个滑动窗口，我们可以将滑动窗口理解为卷积神经网络中的卷积核。然后利用滑动窗口来提取候选区域，通过对候选区域的扫描，会提取出一批特征值或特征区域。接下来通过 SVM、Adaboost 等算法来创建一个分类器，并通过分类器来判断所提取的特征是否属于我们需要进行检测的目标。

在传统的目标检测方法中，一般会使用一些既定的特征提取方法。例如，在人脸检测使用的是一种叫作 Haar 的特征提取方法；而在做其他物体识别的时候，则使用一种叫作 HOG 的特征提取方法；刚性目标检测又会使用一种叫 ACF 的特征提取方法等。不同的特征提取方法，在进行目标检测时的侧重点也是不同的。

例如，在传统的目标检测工程中，人脸检测一般会使用 Adaboost+Haar 的方法。Adaboost 是一种迭代算法，其核心思想是针对同一个训练集训练不同的分类器（弱分类器），然后把这些弱分类器集合起来，构成一个更强的最终分类器（强分类器）。其算法本身是通过改变数据分布来实现的，它根据每次训练集之中每个样本的分类正确情况，以及上次的总体分类的准确率，来确定每个样本的权值。将修改过权值的新数据集送给下层分类器进行训练，最后将每次训练得到的分类器融合起来，作为最后的决策分类器。Adaboost 分类器可以排除一些不必要的训练数据特征，并重点关注关键的训练数据。我们可以通过 Adaboost 方法训练一个弱分类器的组合和级联来完成分类任务。下面来说下具体的做法。

假如针对一幅大小为 $800\text{px} \times 600\text{px}$ 的图像做目标检测，并根据目标检测的结果来确定这幅图像中是否包含人脸。首先我们要定义若干个不同大小的滑动窗口，然后令滑动窗口每次移动 5 个像素，在滑动窗口移动的过程中，将会产生 $800 \times 600 \div 5 = 96\ 000$ 个滑动窗口。而这只是一个滑动窗口大小的滑动所产生的滑动窗口数量，但是在实际的检测过程中，我们会有不同大小的滑动窗口，甚至可能还会对原始图像进行放大处理，此时的总滑动窗口的数量会暴增，而这样的暴增则需要一个效率非常高的分类器。一般来讲，我们使用 Adaboost+cascade 的方法作为此类需求的分类器，通过这种分类器在初期过滤掉一部分候选窗口，再对剩下的难度较大的窗口进行分类判断，直到最后得出判断的结果。

在传统的目标检测中，一般会使用上面的方法进行目标检测，在下一节中我们会介绍在深度学习中是如何进行目标检测的。

14.2 RCNN 介绍

通过 14.1 节的介绍得知，传统的目标检测方法虽然能够解决在实际应用过程中的大部分问题，但是也存在着比较明显的缺陷。因为在传统的目标检测方法中需要设置很多个大大小小的滑动窗口进行特征数据的提取，图像越复杂提取特征所需的滑动窗口数量就会越大，滑动窗口的数量直接影响着目标检测运算的速度，大大地增加了计算量。从另一方面讲，滑动窗口的增多也会导致许多无用窗口的产生，这也会提高复杂度，窗口变得越来越冗余；传统目标检测方法是基于已有的或手工设计的特征进行特征提取的，而对于多样变化的特征来讲并没有很好的鲁棒性，也不利于提升目标检测的效果。

针对这一系列的问题，纽约大学在 2013 年提出了第一个使用深度学习进行目标检测的方法——Overfeat。Overfeat 主要使用卷积神经网络进行训练，首先在分类问题上训练出一个基本模型，再针对这个基本模型进行深入训练，通过不同的视角和尺度来提高分类的置信度，Overfeat 不在原始图片上做滑动窗口，而只在最后一个 pooling 层上做滑动窗口。

在 Overfeat 提出后的第二年，来自加州大学伯克利分校的 Ross Girshick 等人发表了基于卷积神经网络特征的区域方法（Regions with CNN features, RCNN）的文章。RCNN 的诞生可以看成是利用深度学习进行目标检测的一个里程碑。下面开始深入地介绍 RCNN 目标检测方法。

利用 RCNN 进行目标检测，实际上也利用了 CNN 原理。也可以说，RCNN 将 CNN 的价值再一次放大，大大地提升了目标检测的效果，并改变了传统目标检测方法的主要思路。RCNN 实际上是一个基于候选区域的目标检测算法。

RCNN 是利用 CNN 进行目标检测的。但是使用 CNN 进行分类的时候是不需要指定区域或定位的，也就是说，是基于整幅图像进行的，而 RCNN 在进行目标检测的时

候，是在整幅图像中找到需要定位的物体。例如在一张景色照片中找到一个或多个人，而这个人实际上在这张照片上是具有特定的大小和位置的，因此，在进行目标检测时我们需要定位出物体所在的具体位置。也就是说要通过一个框将目标物体给框选出来，并且，在这幅图像中有多少个待检测的物体就要框选多少个。这也是 RCNN 和 CNN 的最大区别。

下面来说一下 RCNN 的具体实现思路。

假定给出一幅候选的图像，使用 RCNN 进行目标检测，首先要在这幅候选图像上生成 1 000~2 000 个候选区域，一般会使用一种叫作 Selective Search 的方法（Selective Search 是一种选择性搜索方法，能够快速找到物体的具体位置）预先提取出一系列比较可能是待识别的物体的候选区域。提取这些候选区域后，会使用 CNN 来提取候选区域上的特征。在提取这些特征的过程中，一般会先将候选区域切分成一个个小的区域。然后对这些小的区域进行“resize”操作，将其变成输入所需要的大小，再针对这些小的区域提取相对高级的特征，并加以存储。这里在切分小区域时，一般会合并可能性最高的两个区域（可能性最高可以理解为颜色相近或纹理相近），然后重复这一步骤，直至整幅图像被合并成一个区域。

提取完特征之后就是进行训练。我们知道，在传统的目标识别方法中一般使用手工标注的特征进行训练，这样的训练方式既浪费了人力资源，标注的准确性还有待商榷。在使用深度学习进行训练时，一般会使用现有的数据集，如 ImageNet 或 PASCAL VOC 2007 等进行分类。因为这些训练数据集都是官方使用大量的图像进行标注的，且经过多次验证，准确性会大大提高。并且，我们可以使用现有的 AlexNet 模型或 VGG16 模型进行训练。研究人员测试结果显示，使用 AlexNet 模型进行训练的准确率约为 58.5%，使用 VGG16 模型训练的准确率约为 66%。虽然使用 VGG16 模型训练出来的精度相对来讲比较高，但是其训练时所消耗的计算量大概是 AlexNet 模型的 7 倍。因此，在一般的目标检测分类训练中，我们建议直接使用 AlexNet 模型进行训练。

RCNN 的训练一般分为三个部分：第一个部分就是上面所讲的使用 AlexNet 模型进行训练。一般来讲，我们将这一轮的训练称为预训练。之所以称之为预训练，是因为还需要对使用这个模型所训练出来的值做进一步调优，也就是说，我们使用 AlexNet

模型进行预训练后所得到的模型的参数实际被用作初始参数，然后使用这个初始参数再进行微调（fine-tuning），得到我们最终想要的结果。

第二部分为调优训练，到目前为止，我们使用 AlexNet 网络模型在 ImageNet 上所输出的神经元个数为 4 096 个，可以将其理解为在训练结束后针对每幅图像都可以得到一个 4 096 维的特征向量。接下来通过微调将这 1 000 个分类转换成 PASCAL VOC 数据集可用的 21 个分类，因为在前面的步骤中已经将现有的模型训练好了，只需要在现有模型的最后将其全连接层的分类数由 1 000 改成 21，并在训练结束后将 f7 层的特征保存即可。

第三部分就是针对上面所训练的每个分类进行类别判断，在这一部分会针对每一类目标使用一个线性的 SVM 分类器进行判断。输入上一个网络所输出的 4 096 维特征，看输出是否属于此类。如果属于待检测的目标，则选框标定，否则认为不属于我们待检测的分类。

在这里需要注意的是，我们使用 SVM 分类器对提取的特征进行打分时，这里的分类数是 20 个，而不是 21 个。因为一般将最后一个作为背景判断的类别，并非待检测目标的类别。

14.3 Fast-RCNN

14.2 节介绍了 RCNN，并了解到使用 RCNN 做目标检测有着非常明显的缺点。RCNN 检测方法需要经过候选区域提取、卷积模型训练、模型微调、分类器训练、边框回归训练等多个步骤，不仅耗时而且占用巨大的空间。因为每张图像需要产生 2 000~3 000 个候选区域，且这些候选区域有很大一部分是重叠的。经过统计，使用 RCNN 进行目标检测训练时，训练 5 000 张图像大约就需要产生几百 GB 的特征文件，这对于一台普通的个人计算机来讲是一个非常大的开销。另外，即使使用 GPU 进行运算，平均处理一幅图像也有 50 秒钟左右的时间，这样的运算速度和运算效率对模型训练来讲是灾难性的。

到了 2014 年，中国的何恺明博士在他的论文“*Spatial Pyramid Pooling in Deep*

Convolutional Networks for Visual Recognition”中提出了一种新型的网络结构——SPPnet。其主要思想就是使用一次卷积来代替 RCNN 中对于候选区域的 2 000 多次卷积操作，将最后提取出来的特征图划分成不同的区域，再将每一个区域的特征图进行卷积操作，将需要提取的区域在原图上的位置映射到卷积层的特征图上。这样的话，针对一幅图像只需要进行一次卷积操作，再将每个需要提取的区域的卷积层特征输入到全连接神经网络中进行全连接操作。另外 SPPnet 采用了一种叫作“空间金字塔池化”的池化层替代了 RCNN 中全连接层之前的最后一个池化层。“空间金字塔池化”是何恺明博士提出的一种池化方法，其目的就是定义一种可伸缩的池化层。也就是说利用“空间金字塔池化”的方法，无论输入的 shape 是什么，最后的输出都是固定的。其原理图如图 14-1 所示。

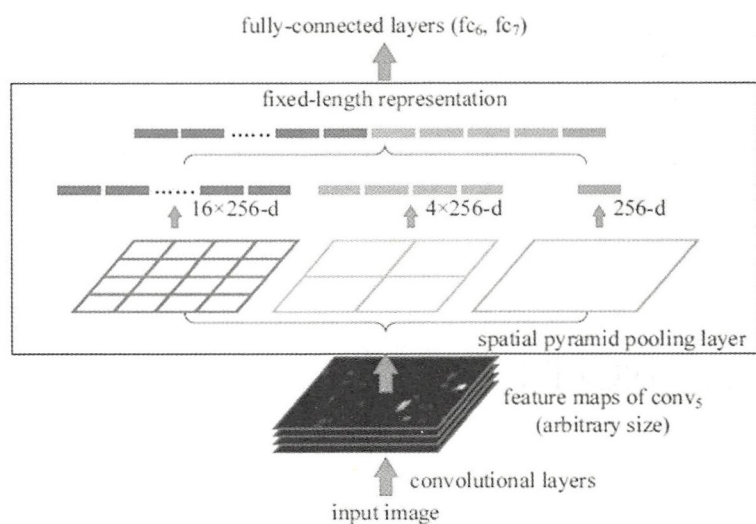


图 14-1

虽然 SPP net 只对原图进行一次卷积操作就解决了 RCNN 的一部分效率问题，但是训练过程还是非常烦琐，要经历候选区域提取、使用 CNN 进行卷积操作提取特征、使用 SVM 进行分类、边框回归训练等多个步骤。另外，仍然没有解决在区域提取过程的耗时问题。

针对 RCNN 和 SPP-net 的缺陷，RCNN 的提出者 Ross Girshick 在 2015 年又提出

了改良版——Fast-RCNN。

Fast-RCNN 一个最重要的特点就是将 RCNN 中深度网络和后面的 SVM 分类两个步骤进行了整合，使用一个全新的网络直接进行分类和回归操作，省去了中间复杂的环节，运算效率得到大幅度的提升。

另外，在 Fast-RCNN 中加入了一个新的层——RoI pooling layer。根据名字我们可以很清楚地知道，这个层实际上也是一个池化层。RoI 是 Region of Interest 的简写，顾名思义，就是在感兴趣的区域上进行框选。我们可以将 RoI 层理解为一个被简化的 SPP 层，使用 RoI 层代替最后一层的最大池化层。在 Fast-RCNN 中，RoI 层实际上是单层的金字塔形，也就是说，只含有一层的池化，因此这可以看作 SPP net 的一个特例，其设计目的主要是满足下一层进行回归和分类时可输入不同大小的特征形状。

RoI 层的作用主要是将图像中的候选区域定位到目标特征中的特征区域，也就是做了一次对应的映射关系，并使用单层的 SPP 将需要传入的目标特征转换为固定大小的特征后传入到全连接层。

另外，在 SPP 网络和 RCNN 网络中，最后一层一般会使用 SVM 算法进行分类。而在 Fast-RCNN 网络中直接将 SVM 分类器去掉，取而代之的是使用 Softmax 函数将整个网络进行整合。那么，为什么要使用 Softmax 函数来取代 SVM 分类器呢？因为在使用 SVM 分类器时，需要将目标特征图存在硬盘上，硬盘数据的读取速度取决于硬盘的种类和性能，因此读取速度会比较慢。而使用 Softmax 函数进行整合，会将目标特征存在显存中，而显存的读取效率会明显高于硬盘的读取效率，因此，使用 Softmax 函数进行整合，会非常明显地提高效率。Fast-RCNN 与 RCNN、SPP net 在速度上的对比如图 14-2 所示。

	Fast - RCNN			RCNN			SPPnet ↑L
	S	M	L	S	M	L	
train time (h)	1.2	2.0	9.5	22	28	84	25
train speedup	18.3×	14.0×	8.8×	1×	1×	1×	3.4×
test rate (s/im)	0.10	0.15	0.32	9.8	12.1	47.0	2.3
▷ with SVD	0.06	0.08	0.22	-	-	-	-
test speedup	98×	80×	146×	1×	1×	1×	20×
▷ with SVD	169×	150×	213×	-	-	-	-
VOC07 mAP	57.1	59.2	66.9	58.5	60.2	66.0	63.1
▷ with SVD	56.5	58.7	66.6	-	-	-	-

图 14-2

另外在做边框回归的时候, Fast-RCNN 采用了一种叫多任务损失(Multi-task Loss) 函数的方法, 将边框回归的训练直接放入到 CNN 中。实际上在这里用了两个输出层, 其中一个输出层是对每个 RoI 输出离散概率的分布, 可以将其理解为分类误差; 另一个输出层则是输出目标边界选框(Bounding Box) 回归的位移, 可以将其理解为定位误差。这两个输出层全部都是采用全连接层来操作的。

Fast-RCNN 的整体结构如图 14-3 所示。

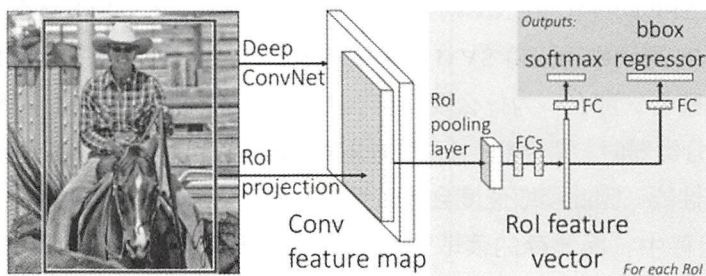


图 14-3

总结起来, Fast-RCNN 的优点主要有:

- (1) 将最后一层的 pooling 改为固定大小的 pooling, 使训练更加方便;
- (2) 使用 Softmax 函数代替了 RCNN 中的 SVM 进行分类, 并在网络中加入了多任务边框回归方法。

14.4 Faster-RCNN

虽然从理论上讲,使用 Fast-RCNN 已经能够比较完美地进行目标检测了,但是实际上还存在着一些潜在的问题。例如,在进行候选特征提取的时候,采用 Selective Search 算法提取一幅图像所消耗的时间大概是 2s,而实际一般都要训练上万幅甚至几十万幅图像。假设训练 10 万幅图像,仅是提取候选区域就要耗时 $100\,000 \times 2 \div 60 \div 24 \approx 2.31$ 天,再加上进行卷积处理和其他操作,仅仅是训练十万幅图像所消耗的时间就相当长。但是在真正的工业领域,我们的数据集远远不止这些,有可能是上千万幅甚至更多,那么其训练速度可想而知。因此,使用 Fast-RCNN 仅能进行小批量数据的检测,真正地放到工业领域进行应用就会显得心有余而力不足。

为了解决这个问题,RCNN 和 Fast-RCNN 的作者 Ross Girshick 基于这两个方法又一次提出了一个更加新的目标检测方法: Faster-RCNN,其网络结构如图 14-4 所示。

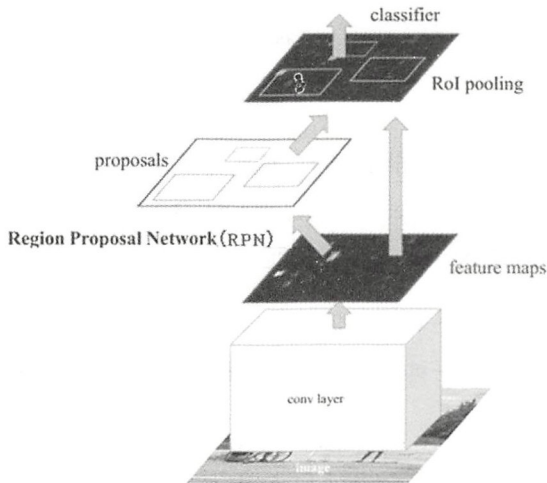


图 14-4

前面所讲的几种目标检测方法都有一个通病,那就是使用 Selective Search 算法作为候选区域提取算法。针对这个问题, Ross Girshick 提出了一种新的算法——EdgeBoxes 算法。EdgeBoxes 算法的优势在于可以将候选区域提取的速度从 2s 缩短到 0.2s,比使用 Selective Search 算法进行候选区域提取速度快了 10 倍。这是一个非常

大的进步。

虽然使用 EdgeBoxes 算法能够大幅度地提升处理图像的速度，但是这个速度对于计算机构建一个神经网络而言仍然是非常慢的。因此，在 Faster-RCNN 中，提出了一个非常重要的网络结构——RPN (Region Proposal Network)，其网络结构图如图 14-5 所示。

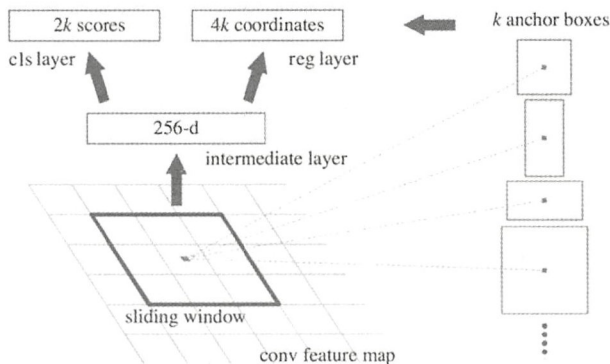


图 14-5

RPN 是一种全卷积神经网络结构，主要用来提取候选框，也就是用来代替前面网络结构中所用到的 Selective Search 算法或 EdgeBoxes 算法。由于没有全连接层，所以可以输入任意大小的图像，而输出是特定大小的。

在卷积神经网络中有一种模型叫作 ZF-Net, ZF-Net 与曾讲过的 AlexNet 模型或者 LetNet-5 模型类似，也是一种非常常用的卷积模型。其网络结构如图 14-6 所示。

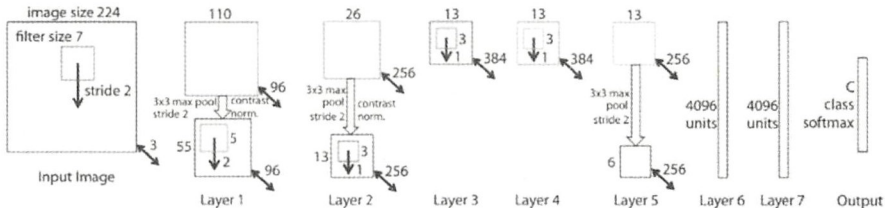


图 14-6

而 RPN 实际上就是基于 ZF-Net 模型的第五层而研究的网络结构。首先根据图 14-6 可以知道,在图 14-5 中 conv feature map 部分的维度为 $13 \times 13 \times 256$ 。然后在这个特征图上,选择一个合适大小的滑窗 (Sliding Window), 比如一个 3×3 的滑窗, 通过滑窗在特征图上进行滑动扫描。滑动扫描的过程实际上就是一个卷积的过程, 卷积之后会产生一个新的低维的全连接特征 (如果采用 ZF-Net 则维度为 256-d, 如果采用 VGG 网络则维度为 512-d)。然后将这个生成的全连接特征送入两个全连接网络中进行运算, 其中一个为分类预测 (a box-classification layer (cls)) 如图 14-5 的左上角部分, 另外一个为边框回归 (a box-regression layer (reg)), 如图 14-5 的右上角部分。我们一般也会将这种同时送入两个全连接网络的方法叫作多任务 (multi-task) 方法。

在图 14-5 的右半部分有一个比较关键的词——anchor (锚), 在这里可以理解为一个锚点, 这个锚点一般位于滑窗的中心处。我们可以将特征图上的每一个点都理解为一个锚点, 而这个锚点的作用实际上就是确定一个选框。在确定特征选框之前将一个点 (例如一个特征的中心点或者一个特征图的左上角) 定为一个锚点, 然后可以根据这个锚点来虚拟地描绘出多个特征。而在图 14-5 中的 k anchor boxes 可以假设有 k 个锚点, 实际上这 k 个锚点可以被理解为 k 个由锚点所产生的选框, 而这些选框产生的作用就是使用 RPN 网络来确定是不是我们所需要的特征物体。我们使用滑动窗口加上 k 个锚点就能够基本覆盖整个图像的区域。锚点还有一个特点就是平移不变性, 也就是说, 如果在图像中将目标物体进行了平移操作, 那么窗口也会跟着平移相同的距离。再回顾下图 14-5 中左上角的分类预测层 (cls layer) 和边框回归层 (reg layer), 在进行分类预测的时候, 会对背景和前景两个类型进行分类预测, 每一个类型可以记 1 分 (score), 那么对于一个区域的预测就会记 2 分, 而目标区域一共有 k 个 anchor boxes, 因此最后所输出的结果是 $2k$ 分; 而对于边框回归 (reg layer) 来讲, 每一个边框会产生 $4k$ 个 Coordinate (坐标), 因此, 最后会产生 $4k$ 个 Coordinates。通过 RPN 模型, 就可以提取出候选区域用于下一个步骤。

在介绍完 RPN 的基本概念和操作后, 我们可以通过图 14-4 的 Faster-RCNN 网络结构很清楚地看到, Faster-RCNN 的网络结构一共被分成了 4 个部分。

第一部分就是 conv layers, 也就是卷积层。在 Faster-RCNN 中, 首先要使用 CNN 的卷积层提取目标图像中的特征图。一般会使用相对比较普遍的提取方式, 即

conv+ReLU 函数+pooling 层。这些特征被提取后会被共享给后续的 RPN 层和全连接层使用。

第二部分就是 RPN，RPN 层在接收 CNN 层提取出来的特征图之后，就会对这些特征图进行目标特征的提取，生成候选区域的选框，并将其传递到下一层。

第三部分就是 RoI Pooling 部分，关于 RoI Pooling 的详细内容我们在 Fast-RCNN 部分已经进行了详细的讲解，其主要作用就是将候选区域进行相应的整理，然后计算出候选区域的特征图并送入到下一层。不过值得注意的是，在 RoI 层中的输入有两个参数：第一个参数是原始的特征（也就是共享的特征），第二个参数是 RPN 所提取出来的候选区域的特征。

第四部分实际上就是进行分类，就是将上一个步骤中已经提取出来的候选区域特征图通过全连接层和 Softmax 函数进行分类计算，以此来确定所框选出来的候选区域到底属于哪一个分类（例如可以是一个人或者一匹马），将每一个分类的概率输出。同时再次利用边框回归的方式来确定更加精确的目标位置并标注，以完成 Faster-RCNN 的整个操作流程。

14.5 YOLO

前面我们所讲的 RCNN、Fast-RCNN、Faster-RCNN 在进行目标检测的过程中实际上都存在着一个共同的特点，那就是都需要分成 2 个大的步骤才能完成对目标的检测。在行业里我们也戏称其为“看两眼”：第一个大的步骤（第一眼）就是提取候选区域；第二个大的步骤（第二眼）就是分类操作。事实上，尽管使用 Faster-RCNN 已经能够很快地进行目标检测了，但是实际上候选区域的提取，仍然是一个比较耗时的操作。那么有没有什么办法能够简化甚至省去这一步呢？针对这个问题，来自华盛顿大学的 Joseph Redmon 和 Ali Farhadi 提出了一个全新的目标检测方法——YOLO。

YOLO 框架出自论文 “*You Only Look Once: Unified, Real-Time Object Detection*”。YOLO（You Only Look Once（你只需要看一次）），相对于之前的 RCNN 系列目标检测方法来讲，最大的特点就是只需要进行一次处理就能够识别出图像中的物体。YOLO

是一个端对端的检测方法，其最大的优势就是速度非常快，因此也被称为实时目标检测。在 RCNN 系列的目标检测方法中所采取的方式是通过滑动窗口的方法进行目标候选区域的提取，然后再送入分类器进行分类识别，但是我们也知道，这类方法存在着速度慢和训练困难的问题。在 YOLO 中将这两个步骤进行合并，将原始图像输入之后，经过一次推理，就能得到图像中所需要识别物体的位置和所属分类，以及相应的置信度。

在前面我们曾详细地讲解了一个叫作 LeNet 的网络，后来，Google 又提出了一个新的网络叫作 GoogLeNet，之所以取名为 GoogLeNet 而不是 GoogleNet，其实也是为了向 LeNet 网络致敬。GoogLeNet 是 2014 年 ILSVRC 挑战赛冠军，它是一个 22 层的深度网络，将 Top5 的错误率降低到 6.67%，而 YOLO 网络实际上就是参照 GoogLeNet 网络而设计的。与 GoogLeNet 网络不同的是，YOLO 没有使用 GoogLeNet 网络中的 Inception module，而是使用了 1×1 的卷积层和 3×3 的卷积层进行了简单的替代。在这里使用 1×1 卷积核的目的是实现跨通道的交互和信息整合，并在保持特征尺寸不变的前提下增加其非线性特性，把网络做得更深。YOLO 网络的检测示意图如图 14-7 所示。

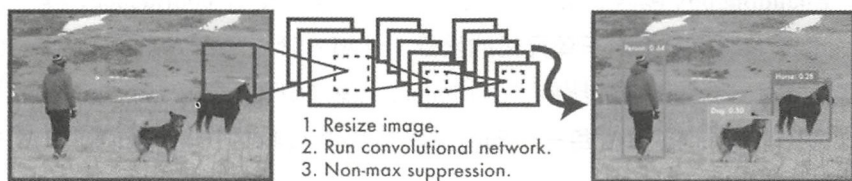


图 14-7

根据图 14-7 可以看出，使用 YOLO 网络进行目标检测时，将一整幅图像输入 YOLO 网络中，然后将其“resize”成一个统一大小的图像，接着将整幅图像放入 CNN 中运行，最后得到目标区域的选框并给出置信度。从宏观角度来讲，YOLO 网络的检测步骤只有这样简简单单的三个步骤，那么，里面的底层原理到底是怎样的呢？

实际上，YOLO 网络在接收图像的输入后，首先会把图像“resize”为一个固定大小的图像，比如 448×448 的图像，再将整幅图像划分成 $S \times S$ 个网格。如果某个物体的中心落在了网格中，也就是说某个网格中包含了某个物体的中心，那么这个网格

就要负责对这个物体进行检测，检测后输出 B 个选框 (Bounding box)，以及 C 个置信度 (也就是属于某种类别物体的概率值)。其基本思路如图 14-8 所示。

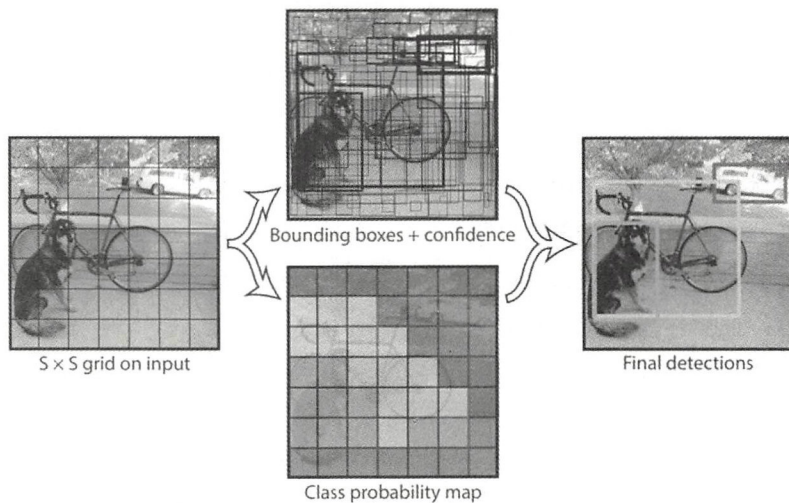


图 14-8

每个 Bounding box 包含 5 个预测值，分别为 x 、 y 、 w 、 h 、confidence，其中 (x, y) 代表 Bounding box 中心相对于单元格的坐标， w 和 h 分别代表 Bounding box 的真实宽度和高度相对于整幅图像的比例，confidence 代表对于每一个 Bounding box 进行预测所得到的置信度，也就是说每一个网格需要预测 $B \times (4+1)$ 个值。我们可以通过以下公式来计算 confidence 的值。

$$\text{confidence} = \Pr(\text{Class}_i | \text{Object}) \times \Pr(\text{Object}) \times \text{IOU}_{\text{pred}}^{\text{truth}} = \Pr(\text{Class}_i) \times \text{IOU}_{\text{pred}}^{\text{truth}}$$

这里的 IOU 表示 Bounding box 与物体交集的面积除以 Bounding box 与物体并集的面积的值， $\Pr(\text{Class}_i | \text{Object})$ 代表每个格子预测出 C 个条件概率，将 Bounding box 的置信度与类别概率做乘法，就可以得到特定类的置信分数，而这个分数也就代表了最后 Bounding box 与目标的吻合程度。如图 14-8 中间部分上面的那幅图像所示，在 YOLO 中，所框选出来的 Bounding box 越粗，表示其置信度越高，反之表示其置信度越低。

假定输入的图像分辨率为 $448\text{px} \times 448\text{px}$, $S=7$, $B=2$, 并且使用 VOC 数据集进行训练, 将其分类设定为 20 类, 因此最终所输出的向量为 $7 \times 7 \times (20+2 \times 5)=1470$ 维, 这里的 $20+2 \times 5$ 是由于每个 Bounding box 包含 5 个预测值并设置 B 为 2 所得到的。其推理过程如图 14-9 所示。

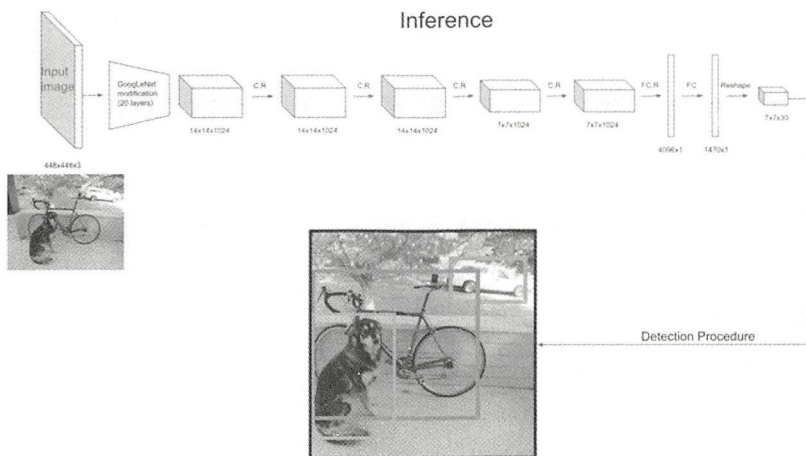


图 14-9

其实这个推理过程就是 YOLO 网络结构实现的过程。我们知道 YOLO 的网络结构借鉴于 GoogLeNet, 并在其基础上进行了改进, YOLO 网络包括 24 个卷积层和 2 个全连接层。YOLO 的网络结构图如图 14-10 所示。

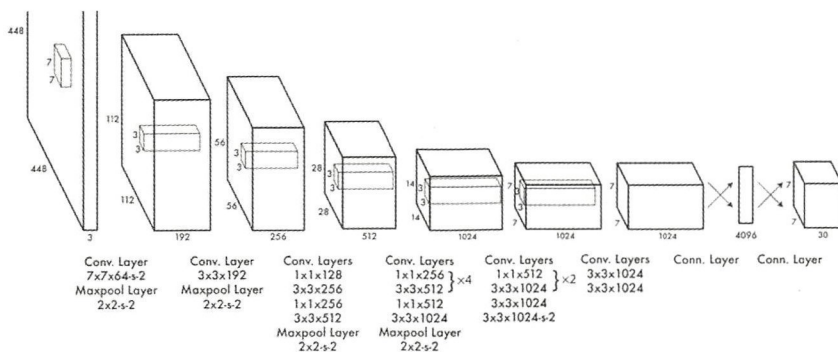


图 14-10

在 YOLO 的网络结构中，我们使用 1×1 的卷积层减少了前一层的特征空间，在一定程度上起到了降维的作用。

在介绍完 YOLO 的基本网络结构及实现过程后，我们来说一下 YOLO 的训练过程。YOLO 的训练一共分为两个大部分：第一部分为预训练，在预训练的过程中一般是使用 ImageNet 数据集进行训练的，训练后得到 1 000 个分类（在训练开始时，将图片的大小“resize”到了 $224\text{px} \times 224\text{px}$ ），预训练主要是用来训练 YOLO 网络的前 20 个卷积层、1 个平均池化层和 1 个全连接层，预训练所得到的模型用作后面最终 YOLO 模型的训练。

第一部分训练结束后，第二部分就是训练真正的 YOLO 模型了。在训练 YOLO 模型时，首先会拿第一部分训练的结果作为输入，即得到的前 20 个卷积层的参数，将这些参数作为第二部分训练的初始化参数，在此基础上进行训练，训练时使用的是 VOC 20 类标注数据。在第二部分中，为了提高训练的精度，我们会将第二部分的输入图像“resize”到 $448\text{px} \times 448\text{px}$ 。

在训练 YOLO 模型的过程中，损失函数的定义是非常关键的部分。

$$\begin{aligned} & \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B I_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\ & + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B I_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\ & + \sum_{i=0}^{S^2} \sum_{j=0}^B I_{ij}^{\text{obj}} (C_i - \hat{C}_i)^2 \\ & + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^B I_{ij}^{\text{noobj}} (C_i - \hat{C}_i) \\ & + \sum_{j=0}^B I_{ij}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2 \end{aligned}$$

这个损失函数主要包含四个部分，首先是前面两行：

$$\begin{aligned} & \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B I_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\ & + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B I_{ij}^{\text{obj}} \left[(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \right] \end{aligned}$$

这段公式主要的目的就是进行坐标预测，其中 I_{ij}^{obj} 是判断第 i 个网格中的第 j 个 Bounding box 是否负责这个 object。在第一行中，使用坐标 x 和 y 的误差平方和来预测 Bounding box 的中心坐标，而第二行则是对 Bounding box 的宽度和高度进行预测。接下来看第三行和第四行：

$$\begin{aligned} & + \sum_{i=0}^{S^2} \sum_{j=0}^B I_{ij}^{\text{obj}} (C_i - \hat{C}_i)^2 \\ & + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^B I_{ij}^{\text{noobj}} (C_i - \hat{C}_i)^2 \end{aligned}$$

这两行的目的主要是计算 confidence，可以发现，这两行和上面两行有一个明显的区别就是将 I_{ij}^{obj} 替换成了 I_{ij}^{noobj} ，实际上这两个公式的确意思是相反的， I_{ij}^{noobj} 的含义是用来判断是否包含 object，如果不包含 object，那么 confidence 就会被设置为 0，如果包含 object，那么 confidence 就会被设置为 1。在这两行中，上面一行代表包含 object 的 confidence 值的预测，而下面一行代表不包含 object 的 confidence 值的预测。接下来我们再看最后一行：

$$+ \sum_{j=0}^B I_{ij}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2$$

最后一行就是针对前面所有的关于 confidence 值和 Bounding box 而做的最终的类别预测，其中 I_{ij}^{obj} 用来判断是否有 object 的中心落在网格 i 中，如果落在了网格 i 中，则这个网格就负责预测这个 object 的概率。

对于整个损失函数，有几个点需要注意。首先就是关于 λ_{coord} 和 λ_{noobj} 的定义。这两个值都是代表损失权重 (Loss Weight)，其中 λ_{coord} 是针对 8 维的坐标预测，给这些损失前面赋予更大的损失权重，在 pascal VOC 上进行训练时取 5； λ_{noobj} 是针对没有 object



的 Bounding box 的置信度损失值，会给这些损失前面赋予更小的损失权重，在 pascal VOC 上进行训练时取 0.5；对于有 object 的 Bounding box 的置信度损失和类别损失一般会统一取值 1.0。

损失函数设计的目的就是让坐标 (x,y,w,h)、置信度、分类这三个方面达到一个相对完美的平衡，并简单地采用了 sum-squared error loss 的方式来实现这个效果。

损失函数在训练 YOLO 的过程中起到了目标检测和分类的效果。在训练 YOLO 时，损失函数只在当前网格中含有 object 时才考虑其分类的错误误差。

刚刚讲解了 YOLO 的训练过程，接下来我们就使用 TensorFlow 来进行 YOLO 模型的训练和识别。

使用 TensorFlow 训练 YOLO 模型时同样遵循着上面所说的两个步骤，即预训练模型和训练最终的 YOLO 模型。取得 YOLO 预训练模型的方法一般有两种：第一种是使用 CNN 来自己训练 ImageNet 网络，然后取其前 20 层，生成一个预训练模型；第二种则是使用 YOLO 官方所提供的预训练模型，并在这个预训练模型的基础上进行训练，来取得最终的 YOLO 模型。一般来讲，我们比较推荐第二种方法。

因为在使用 ImageNet 训练 YOLO 的预训练模型时，我们使用的方法及函数和官方所使用的几乎是一样的，从 0 开始训练一个模型需要耗费很长的时间，并且在这么长的训练之后所得出的效果一般和官方所得出的效果是差不多的，完全没有必要从头开始训练这个预训练模型，所以推荐使用官方训练好的模型。在此基础上进行二次训练来得到我们想要的结果模型。接下来就讲解一下如何使用 TensorFlow 框架在官方模型的基础上进行训练。

在训练 YOLO 模型之前，先要准备进行训练的数据，这里使用的是 VOC 2007 数据集的数据，使用 `wget http://pjreddie.com/media/files/VOCtrainval_06-Nov-2007.tar` 命令进行下载（注意：这里下载的数据集为 VOC 训练数据集）。其次我们需要准备预训练的模型，YOLO 已经为我们准备好了预训练的模型，可以直接使用它。一般在 TensorFlow 下常用的 YOLO 模型为 `yolo_tiny.ckpt` 或 `YOLO_small.ckpt`。本例中我们使用 `YOLO_small.ckpt` 作为 TensorFlow 的基础训练模型进行二次训练开发（读者可以很



容易地从搜索引擎中找到)。

下面来说一说如何使用 TensorFlow 训练一个 YOLO 模型并加以验证。

训练一个 YOLO 模型主要经历以下步骤：

- (1) 读取参数；
- (2) 训练 YOLONet 模型；
- (3) 读取训练数据；
- (4) 训练并验证。

那么第一步，我们先来看下如何定义 YOLONet 模型。定义一个 YOLONet 模型首先定义配置文件 config.py，config.py 文件主要包含了定义路径、数据集参数、模型参数、求解参数和测试参数这五个部分。数据集参数主要规定了训练所用的数据集的路径、分类和权重等信息，代码如下：

```
DATA_PATH = 'data'
PASCAL_PATH = os.path.join(DATA_PATH, 'pascal_voc')
CACHE_PATH = os.path.join(PASCAL_PATH, 'cache')
OUTPUT_DIR = os.path.join(PASCAL_PATH, 'output')
WEIGHTS_DIR = os.path.join(PASCAL_PATH, 'weights')
WEIGHTS_FILE = None
# WEIGHTS_FILE = os.path.join(DATA_PATH, 'weights', 'YOLO_small.ckpt')
CLASSES = ['aeroplane', 'bicycle', 'bird', 'boat', 'bottle', 'bus',
            'car', 'cat', 'chair', 'cow', 'diningtable', 'dog', 'horse',
            'motorbike', 'person', 'pottedplant', 'sheep', 'sofa',
            'train', 'tvmonitor']

FLIPPED = True
```

在这段代码中，我们可以看到有一句被注释掉的代码# WEIGHTS_FILE = os.path.join(DATA_PATH, 'weights', 'YOLO_small.ckpt')，这句代码之所以被注释掉，主要是因为对于权重文件，我们可以在配置文件中定义，也可以在代码中定义。如果确定好就是使用这个模型的权重文件的话，那么这里可以写“死”。但是一般来讲，我们可能会频繁地使用自己训练的分类模型进行加载，所以在这里把这句代码注释掉，



等到训练时在代码中加入即可。

另外可以看到，在这里的 CLASSES 中定义了 20 个分类，这 20 个分类也就是我们要训练的分类模型。在最后一句中，定义了 FLIPPED = True，这句话的意思是数据集的数据可以被翻转使用。

定义好数据集的相关参数后，接下来就要定义模型的相关参数。模型的相关参数一般包括图像的大小、网格的大小等，定义代码如下：

```
IMAGE_SIZE = 448
CELL_SIZE = 7
BOXES_PER_CELL = 2
ALPHA = 0.1
DISP_CONSOLE = False
OBJECT_SCALE = 1.0
NOOBJECT_SCALE = 1.0
CLASS_SCALE = 2.0
COORD_SCALE = 5.0
```

从上述代码可以看出，我们定义了图像的大小为 448px，这里的 448px 可以理解为输入图像的大小会被“resize”为 448px × 448px，然后规定网格大小为 7 × 7，每个网格会产生两个 Bounding box 等，其实这里面的参数就是我们在前面讲解 YOLO 模型时所用到的参数。

接下来定义求解参数，求解参数定义的目的是训练网络模型，后续会在训练文件 train.py 中用得到，代码如下：

```
GPU = ''
LEARNING_RATE = 0.0001
DECAY_STEPS = 30000
DECAY_RATE = 0.1
STAIRCASE = True
BATCH_SIZE = 45
MAX_ITER = 15000
SUMMARY_ITER = 10
SAVE_ITER = 1000
```

在这里，设置了学习率为 0.0001，衰变步数为 30 000 步，每 1 000 步保存一次等。具体使用在讲解训练代码时会着重讲解。接下来，设置测试的相关参数，测试参数主



要是在进行模型测试时需要用到的，测试参数主要设置了相关的阈值。代码如下：

```
THRESHOLD = 0.2
IOU_THRESHOLD = 0.5
```

将上面的所有代码部分统一起来，就是 `config.py` 文件，这个文件也是所有配置文件的核心。之后训练、计算所涉及的一些相对固定的配置参数都会从这个文件中调用和读取。

在了解了配置文件后，接下来要做的事情就是进行 YOLO 网络的结构初始化，结构初始化保存在 `yolo_net.py` 文件中。初始化 YOLO 网络同样也分为几个步骤：

- (1) 初始化参数；
- (2) 构建网络；
- (3) 计算 IOU；
- (4) 定义损失层；
- (5) 激活网络。

首先来看一下如何初始化参数，刚刚已经定义了基本的网络参数，主要应用之前所定义好的参数进行网络的初始化。代码如下：

```
import yolo.config as cfg

def __init__(self, is_training=True):
    self.classes = cfg.CLASSES
    self.num_class = len(self.classes)
    self.image_size = cfg.IMAGE_SIZE
    self.cell_size = cfg.CELL_SIZE
    self.bboxes_per_cell = cfg.BOXES_PER_CELL
    self.output_size = (self.cell_size * self.cell_size) * \
        (self.num_class + self.bboxes_per_cell * 5)
    self.scale = 1.0 * self.image_size / self.cell_size
    self.boundary1 = self.cell_size * self.cell_size * self.num_class
    self.boundary2 = self.boundary1 + \
        self.cell_size * self.cell_size * self.bboxes_per_cell

    self.object_scale = cfg.OBJECT_SCALE
```




```
self.noobject_scale = cfg.NOOBJECT_SCALE
self.class_scale = cfg.CLASS_SCALE
self.coord_scale = cfg.COORD_SCALE

self.learning_rate = cfg.LEARNING_RATE
self.batch_size = cfg.BATCH_SIZE
self.alpha = cfg.ALPHA

self.offset = np.transpose(np.reshape(np.array(
    [np.arange(self.cell_size)] * self.cell_size * self.bboxes_per_cell),
    (self.bboxes_per_cell, self.cell_size, self.cell_size)), (1, 2, 0))

self.images = tf.placeholder(
    tf.float32, [None, self.image_size, self.image_size, 3],
    name='images')
self.logist = self.build_network(
    self.images, num_outputs=self.output_size, alpha=self.alpha,
    is_training=is_training)

if is_training:
    self.labels = tf.placeholder(
        tf.float32,
        [None, self.cell_size, self.cell_size, 5 + self.num_class])
    self.loss_layer(self.logist, self.labels)
    self.total_loss = tf.losses.get_total_loss()
    tf.summary.scalar('total_loss', self.total_loss)
```

导入配置文件 `import yolo.config as cfg` 后,就开始进行参数的设置。在定义 YOLO 网络的过程中,前半部分主要应用了刚刚定义的一些基本参数,例如分类信息、图像大小、网格大小等,接下来主要讲解在 `config.py` 文件中没有涉及的部分。

其中参数 `output_size` 指的是网络最后输出的尺寸大小,这个尺寸大小实际上是经过计算而来的,使用每一个网格的大小(也就是 7×7),再乘以分类数加上 2 个 Bounding box,每个 Bounding box 又有 5 个输出,因此最后所得到的值为 $(self.cell_size \times self.cell_size) \times (self.num_class + self.bboxes_per_cell \times 5)$,也就是我们在前面所讲解的输出的 1470 维的向量。

`scale` 可以理解为输出的网格规模,计算方法是用每一个图像的尺寸除以网格的大小,也就是 $448 \div 7 = 64$,当然,这只是一行的尺寸。



boundary1 和 boundary2 代表两个边框的大小：第一个边框是用网格的尺寸乘以分类数，即 $7 \times 7 \times 20$ ；第二个边框是在第一个边框的基础上加上网格的尺寸乘以每一个边框的大小，即 $7 \times 7 \times 20 + 7 \times 7 \times 2$ 。

offset 参数是指偏置量的计算，在操作中调用了 `np.transpose()` 函数去求矩阵的转置，而实际上利用了 `np.reshape` 函数重新定义了张量的阶数。在 TensorFlow 中是利用张量来表示数据的，张量是有阶数的，一阶张量实际上就是一个数组，二阶张量是一个矩阵。而 `shape` 实际上就是张量的形状，因此 `offset` 的计算实际上就是利用矩阵的转置而求得的。

`is_training` 函数实际上就是将这些训练的参数传入以方便后续的计算。

当完成网络的初始化后，接下来要做的就是构建 YOLO 网络结构，先来看下构建部分的代码：

```
def build_network(self,
                    images,
                    num_outputs,
                    alpha,
                    keep_prob=0.5,
                    is_training=True,
                    scope='yolo'):
    with tf.variable_scope(scope):
        with slim.arg_scope(
            [slim.conv2d, slim.fully_connected],
            activation_fn=leaky_relu(alpha),
            weights_regularizer=slim.l2_regularizer(0.0005),
            weights_initializer=tf.truncated_normal_initializer(0.0, 0.01)
        ):
            net = tf.pad(
                images, np.array([[0, 0], [3, 3], [3, 3], [0, 0]]),
                name='pad_1')
            net = slim.conv2d(
                net, 64, 7, 2, padding='VALID', scope='conv_2')
            net = slim.max_pool2d(net, 2, padding='SAME', scope='pool_3')
            net = slim.conv2d(net, 192, 3, scope='conv_4')
            net = slim.max_pool2d(net, 2, padding='SAME', scope='pool_5')
            net = slim.conv2d(net, 128, 1, scope='conv_6')
            net = slim.conv2d(net, 256, 3, scope='conv_7')
```



```

net = slim.conv2d(net, 256, 1, scope='conv_8')
net = slim.conv2d(net, 512, 3, scope='conv_9')
net = slim.max_pool2d(net, 2, padding='SAME', scope='pool_10')
net = slim.conv2d(net, 256, 1, scope='conv_11')
net = slim.conv2d(net, 512, 3, scope='conv_12')
net = slim.conv2d(net, 256, 1, scope='conv_13')
net = slim.conv2d(net, 512, 3, scope='conv_14')
net = slim.conv2d(net, 256, 1, scope='conv_15')
net = slim.conv2d(net, 512, 3, scope='conv_16')
net = slim.conv2d(net, 256, 1, scope='conv_17')
net = slim.conv2d(net, 512, 3, scope='conv_18')
net = slim.conv2d(net, 512, 1, scope='conv_19')
net = slim.conv2d(net, 1024, 3, scope='conv_20')
net = slim.max_pool2d(net, 2, padding='SAME', scope='pool_21')
net = slim.conv2d(net, 512, 1, scope='conv_22')
net = slim.conv2d(net, 1024, 3, scope='conv_23')
net = slim.conv2d(net, 512, 1, scope='conv_24')
net = slim.conv2d(net, 1024, 3, scope='conv_25')
net = slim.conv2d(net, 1024, 3, scope='conv_26')
net = tf.pad(
    net, np.array([[0, 0], [1, 1], [1, 1], [0, 0]]),
    name='pad_27')
net = slim.conv2d(
    net, 1024, 3, 2, padding='VALID', scope='conv_28')
net = slim.conv2d(net, 1024, 3, scope='conv_29')
net = slim.conv2d(net, 1024, 3, scope='conv_30')
net = tf.transpose(net, [0, 3, 1, 2], name='trans_31')
net = slim.flatten(net, scope='flat_32')
net = slim.fully_connected(net, 512, scope='fc_33')
net = slim.fully_connected(net, 4096, scope='fc_34')
net = slim.dropout(
    net, keep_prob=keep_prob, is_training=is_training,
    scope='dropout_35')
net = slim.fully_connected(
    net, num_outputs, activation_fn=None, scope='fc_36')
return net

```

通过代码很容易发现，其实构建网络函数 `build_network` 所做的工作就是构建 YOLO 的卷积网络和全连接网络。网络中的参数实际上就是我们在前面讲解的网络结构图中的各种输入，最后使用 `dropout` 函数来防止过拟合现象的发生，并使用一个全连接层进行网络的组合，最后返回整个网络。网络的构建方式可以参考 YOLO 网络结

构和图 14-10。

在构建完基本的网络结构之后，接下来要做的就是处理并计算 IOU 层，以及计算损失函数，代码如下：

```
def calc_iou(self, boxes1, boxes2, scope='iou'):
    with tf.variable_scope(scope):
        boxes1_t = tf.stack([boxes1[..., 0] - boxes1[..., 2] / 2.0,
                             boxes1[..., 1] - boxes1[..., 3] / 2.0,
                             boxes1[..., 0] + boxes1[..., 2] / 2.0,
                             boxes1[..., 1] + boxes1[..., 3] / 2.0],
                             axis=-1)
        boxes2_t = tf.stack([boxes2[..., 0] - boxes2[..., 2] / 2.0,
                             boxes2[..., 1] - boxes2[..., 3] / 2.0,
                             boxes2[..., 0] + boxes2[..., 2] / 2.0,
                             boxes2[..., 1] + boxes2[..., 3] / 2.0],
                             axis=-1)

        lu = tf.maximum(boxes1_t[..., :2], boxes2_t[..., :2])
        rd = tf.minimum(boxes1_t[..., 2:], boxes2_t[..., 2:])

        intersection = tf.maximum(0.0, rd - lu)
        inter_square = intersection[..., 0] * intersection[..., 1]

        square1 = boxes1_t[..., 2] * boxes1_t[..., 3]
        square2 = boxes2_t[..., 2] * boxes2_t[..., 3]
        union_square = tf.maximum(square1 + square2 - inter_square, 1e-10)
        return tf.clip_by_value(inter_square / union_square, 0.0, 1.0)

def loss_layer(self, predicts, labels, scope='loss_layer'):
    with tf.variable_scope(scope):
        predict_classes = tf.reshape(
            predicts[:, :self.boundary1],
            [self.batch_size, self.cell_size, self.cell_size, self.num_class])
        predict_scales = tf.reshape(
            predicts[:, self.boundary1:self.boundary2],
            [self.batch_size, self.cell_size, self.cell_size,
             self.bboxes_per_cell])
        predict_boxes = tf.reshape(
            predicts[:, self.boundary2:],
            [self.batch_size, self.cell_size, self.cell_size,
             self.bboxes_per_cell, 4])
```



```

response = tf.reshape(
    labels[..., 0],
    [self.batch_size, self.cell_size, self.cell_size, 1])
boxes = tf.reshape(
    labels[..., 1:5],
    [self.batch_size, self.cell_size, self.cell_size, 1, 4])
boxes = tf.tile(
    boxes, [1, 1, 1, self.boxes_per_cell, 1]) / self.image_size
classes = labels[..., 5:]
offset = tf.reshape(
    tf.constant(self.offset, dtype=tf.float32),
    [1, self.cell_size, self.cell_size, self.boxes_per_cell])
offset = tf.tile(offset, [self.batch_size, 1, 1, 1])
offset_tran = tf.transpose(offset, (0, 2, 1, 3))
predict_boxes_tran = tf.stack(
    [(predict_boxes[..., 0] + offset) / self.cell_size,
     (predict_boxes[..., 1] + offset_tran) / self.cell_size,
     tf.square(predict_boxes[..., 2]),
     tf.square(predict_boxes[..., 3])], axis=-1)
iou_predict_truth = self.calc_iou(predict_boxes_tran, boxes)

object_mask = tf.reduce_max(iou_predict_truth, 3, keep_dims=True)
object_mask = tf.cast(
    (iou_predict_truth >= object_mask), tf.float32) * response

noobject_mask = tf.ones_like(
    object_mask, dtype=tf.float32) - object_mask

boxes_tran = tf.stack(
    [boxes[..., 0] * self.cell_size - offset,
     boxes[..., 1] * self.cell_size - offset_tran,
     tf.sqrt(boxes[..., 2]),
     tf.sqrt(boxes[..., 3])], axis=-1)

class_delta = response * (predict_classes - classes)
class_loss = tf.reduce_mean(
    tf.reduce_sum(tf.square(class_delta), axis=[1, 2, 3]),
    name='class_loss') * self.class_scale

object_delta = object_mask * (predict_scales - iou_predict_truth)
object_loss = tf.reduce_mean(
    tf.reduce_sum(tf.square(object_delta), axis=[1, 2, 3]),
    name='object_loss') * self.object_scale

```

```
noobject_delta = noobject_mask * predict_scales
noobject_loss = tf.reduce_mean(
    tf.reduce_sum(tf.square(noobject_delta), axis=[1, 2, 3]),
    name='noobject_loss') * self.noobject_scale

# coord_loss
coord_mask = tf.expand_dims(object_mask, 4)
boxes_delta = coord_mask * (predict_boxes - boxes_tran)
coord_loss = tf.reduce_mean(
    tf.reduce_sum(tf.square(boxes_delta), axis=[1, 2, 3, 4]),
    name='coord_loss') * self.coord_scale

tf.losses.add_loss(class_loss)
tf.losses.add_loss(object_loss)
tf.losses.add_loss(noobject_loss)
tf.losses.add_loss(coord_loss)

tf.summary.scalar('class_loss', class_loss)
tf.summary.scalar('object_loss', object_loss)
tf.summary.scalar('noobject_loss', noobject_loss)
tf.summary.scalar('coord_loss', coord_loss)

tf.summary.histogram('boxes_delta_x', boxes_delta[..., 0])
tf.summary.histogram('boxes_delta_y', boxes_delta[..., 1])
tf.summary.histogram('boxes_delta_w', boxes_delta[..., 2])
tf.summary.histogram('boxes_delta_h', boxes_delta[..., 3])
tf.summary.histogram('iou', iou_predict_truth)
```

在计算 IOU 层的时候，首先要做的就是进行矩阵拼接。通过调用 `tf.stack()` 函数将两个 Bounding box 的矩阵进行拼接，形成一个新的矩阵。因为原始的输入实际上是一个 4-D 的张量 (`[CELL_SIZE, CELL_SIZE, BOXES_PER_CELL, 4]`)，需要将其转换为 `(x_center, y_center, w, h)` 的形式。因此采用这种拼接转换的方法来求出两个 boxes 的值。

`lu` 和 `rd` 实际上是 `left up` 和 `right down` 的简称，顾名思义，实际上就是寻找左上角和右下角的两个点，也就是计算位置信息。计算位置信息的过程实际上很简单，仅使用 `tf.maximum` 和 `tf.minimum` 来寻找 Bounding Box 中的最大坐标和最小坐标即可实现。

损失函数被定义在 `loss_layer(self, predicts, labels, scope='loss_layer')` 方法中，其内部定义的参数和我们前面所讲解的损失函数的公式定义是一样的，这里仅对一些比较生疏的变量做一下解释。

`object_mask` 是指计算包含 IOU 部分的张量形状 `[BATCH_SIZE, CELL_SIZE, CELL_SIZE, BOXES_PER_CELL]`，主要通过 `tf.reduce_max()` 函数找出各行中的最大值组成一个张量（Tensor），并输出为指定的形状。而 `noobject_mask` 是指不包含目标的张量大小，通过 `tf.ones_like()` 函数将 `object_mask` 的形状进行复制，并使用全部为 1 的值来填满，再减去 `object_mask` 获得最终值。

`class_loss`、`noobject_loss`、`coord_loss` 主要用来计算各种损失函数，其计算方式可以参考 `loss` 的定义公式。

最后将值进行保存，以待训练时使用。

在网络的最后，定义一个激活函数 `leaky_relu` 用于激活，代码如下：

```
def leaky_relu(alpha):  
    def op(inputs):  
        return tf.nn.leaky_relu(inputs, alpha=alpha, name='leaky_relu')  
    return op
```

到目前为止，我们训练之前所有的准备工作都已经完成，接下来要做的就是编写 `train.py` 文件进行模型的训练。由于有了之前各种定义，所以在训练过程中主要使用之前定义好的各种参数和方法。训练主要包含以下几个部分：

- （1）初始化网络参数；
- （2）训练模型；
- （3）保存模型并更新参数信息。

首先来说一下初始化网络参数，其实在训练 YOLO 模型的时候，所需要初始化的网络参数一部分在配置文件中已经被定义，另外一部分则需要在训练前自己来定义。那么首先来看一下初始化参数的代码：

```
def __init__(self, net, data):
```



```
self.net = net
self.data = data
self.weights_file = cfg.WEIGHTS_FILE
self.max_iter = cfg.MAX_ITER
self.initial_learning_rate = cfg.LEARNING_RATE
self.decay_steps = cfg.DECAY_STEPS
self.decay_rate = cfg.DECAY_RATE
self.staircase = cfg.STAIRCASE
self.summary_iter = cfg.SUMMARY_ITER
self.save_iter = cfg.SAVE_ITER
self.output_dir = os.path.join(
    cfg.OUTPUT_DIR, datetime.datetime.now().strftime('%Y_%m_%d_%H_%M'))
if not os.path.exists(self.output_dir):
    os.makedirs(self.output_dir)
self.save_cfg()
self.variable_to_restore = tf.global_variables()
self.saver = tf.train.Saver(self.variable_to_restore, max_to_keep=None)
self.ckpt_file = os.path.join(self.output_dir, 'yolo')
self.summary_op = tf.summary.merge_all()
self.writer = tf.summary.FileWriter(self.output_dir, flush_secs=60)

self.global_step = tf.train.create_global_step()
self.learning_rate = tf.train.exponential_decay(
    self.initial_learning_rate, self.global_step, self.decay_steps,
    self.decay_rate, self.staircase, name='learning_rate')
self.optimizer = tf.train.GradientDescentOptimizer(
    learning_rate=self.learning_rate)
self.train_op = slim.learning.create_train_op(
    self.net.total_loss, self.optimizer, global_step=self.global_step)
gpu_options = tf.GPUOptions()
config = tf.ConfigProto(gpu_options=gpu_options)
self.sess = tf.Session(config=config)
self.sess.run(tf.global_variables_initializer())
if self.weights_file is not None:
    print('Restoring weights from: ' + self.weights_file)
    self.saver.restore(self.sess, self.weights_file)
self.writer.add_graph(self.sess.graph)
```

初始化参数代码的前半部分用的都是 config.py 这个配置文件中的配置信息，这里主要讲一下重新定义的信息，以及比较难理解的信息。

output_dir 参数是以时间命名的一个文件夹，也就是每次在训练的开始都会以当

前的时间为名称建立文件夹，这个文件夹的主要作用就是保存训练的结果和参数信息。

ckpt_file 参数主要是在 output_dir 文件夹下面加入 YOLO 的 ckpt 文件信息，一般来讲会以这个 ckpt 文件为名称（在这里是 ‘yolo’）保存。这里要注意的是，在这里所保存的并不是一个 ckpt 实体文件，而是相关的网络参数，例如 meta、index 等信息。

writer 参数主要把模型中的参数保存到 output_dir 变量所在的文件夹下。

global_step 和 learning_rate 主要用于监控训练过程中的总步数和学习率。

```
if self.weights_file is not None:
    print('Restoring weights from: ' + self.weights_file)
    self.saver.restore(self.sess, self.weights_file)

self.writer.add_graph(self.sess.graph)
```

这段代码的意思是将 weight 文件下的权重参数恢复出来，并将参数添加到 session 中（这里使用的是 “YOLO_small.ckpt” 文件，在训练时我们会讲这个部分）。

接下来就开始训练，训练部分代码如下：

```
def train(self):

    train_timer = Timer()
    load_timer = Timer()

    for step in range(1, self.max_iter + 1):

        load_timer.tic()
        images, labels = self.data.get()
        load_timer.toc()
        feed_dict = {self.net.images: images,
                     self.net.labels: labels}

        if step % self.summary_iter == 0:
            if step % (self.summary_iter * 10) == 0:

                train_timer.tic()
                summary_str, loss, _ = self.sess.run(
                    [self.summary_op, self.net.total_loss, self.train_op],
                    feed_dict=feed_dict)
```

```

        train_timer.toc()

        log_str = '{} Epoch: {}, Step: {}, Learning rate: {}, ''
        '' Loss: {:.3f}\nSpeed: {:.3f}s/iter, ''
        '' Load: {:.3f}s/iter, Remain: {}'.format(
            datetime.datetime.now().strftime('%m-%d %H:%M:%S'),
            self.data.epoch,
            int(step),
            round(self.learning_rate.eval(session=self.sess), 6),
            loss,
            train_timer.average_time,
            load_timer.average_time,
            train_timer.remain(step, self.max_iter))
        print(log_str)

    else:
        train_timer.tic()
        summary_str, _ = self.sess.run(
            [self.summary_op, self.train_op],
            feed_dict=feed_dict)
        train_timer.toc()

        self.writer.add_summary(summary_str, step)

    else:
        train_timer.tic()
        self.sess.run(self.train_op, feed_dict=feed_dict)
        train_timer.toc()

    if step % self.save_iter == 0:
        print('{} Saving checkpoint file to: {}'.format(
            datetime.datetime.now().strftime('%m-%d %H:%M:%S'),
            self.output_dir))
        self.saver.save(
            self.sess, self.ckpt_file, global_step=self.global_step)

def save_cfg(self):
    with open(os.path.join(self.output_dir, 'config.txt'), 'w') as f:
        cfg_dict = cfg.__dict__
        for key in sorted(cfg_dict.keys()):
            if key[0].isupper():
                cfg_str = '{}: {}\n'.format(key, cfg_dict[key])
                f.write(cfg_str)

```

其实训练部分的代码逻辑结构是非常清晰的，首先是通过 `self.data.get()` 函数从数据集中获取数据，这里面的 `data` 实际上是在 `main` 函数中通过数据集进行传递的。在 `main` 函数中会有 `pascal = pascal_voc('train')` 这么一句话，这句话的作用就是从 VOC 数据集中获取数据，然后赋值给 `pascal` 变量。而这个 `pascal` 是一个被定义好的专门用来读取数据和处理数据的文件。

在读取数据之后，我们通过 `feed_dict` 命令将数据“喂”进去，然后就开始一步步地进行迭代训练，并且每隔 100 个 `step` 就用 `log_str` 命令输出一一次 `log` 值，这个 `log_str` 命令包括了训练的时间、步骤、损失、速度等信息。当全部训练完成后，调用 `saver.save` 方法保存模型，并使用 `save_cfg` 函数保存相关参数。

最后再来说一下更新配置函数和主函数：

```
def update_config_paths(data_dir, weights_file):
    cfg.DATA_PATH = data_dir
    cfg.PASCAL_PATH = os.path.join(data_dir, 'pascal_voc')
    cfg.CACHE_PATH = os.path.join(cfg.PASCAL_PATH, 'cache')
    cfg.OUTPUT_DIR = os.path.join(cfg.PASCAL_PATH, 'output')
    cfg.WEIGHTS_DIR = os.path.join(cfg.PASCAL_PATH, 'weights')

    cfg.WEIGHTS_FILE = os.path.join(cfg.WEIGHTS_DIR, weights_file)

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--weights', default="YOLO_small.ckpt", type=str)
    parser.add_argument('--data_dir', default="data", type=str)
    parser.add_argument('--threshold', default=0.2, type=float)
    parser.add_argument('--iou_threshold', default=0.5, type=float)
    parser.add_argument('--gpu', default='', type=str)
    args = parser.parse_args()
    if args.gpu is not None:
        cfg.GPU = args.gpu
    if args.data_dir != cfg.DATA_PATH:
        update_config_paths(args.data_dir, args.weights)
    os.environ['CUDA_VISIBLE_DEVICES'] = cfg.GPU
    yolo = YOLONet()
    pascal = pascal_voc('train')
    solver = Solver(yolo, pascal)
    print('Start training ...')
    solver.train()
```



```
print('Done training.')
if __name__ == '__main__':
    # python train.py --weights YOLO_small.ckpt --gpu 0
main()
```

首先来看一下 main 函数，在 main 函数中，第一步要做的就是设置相关参数，这些参数包括加载的预训练模型权重、使用的 YOLO_small.ckpt 文件、设置数据存放路径和使用的 GPU 等。

YOLO 表示使用的是 YOLONet() 这个文件里面的模型，这个文件在代码的一开始就讲过，在 main 函数中，设置完相关参数后，就调用了 train() 函数开始相关的训练并得到最后的模型。

至此整个 YOLO 的训练就完成了。在 YOLO 训练的过程中有几个比较重要的步骤。首先采用 P5000 的 GPU 进行训练，训练时间大概在 5 个小时左右，每隔 1 000 步会保存一次训练的模型，保存后，会在项目的 data 目录下生成一个以时间为命名的文件夹，文件夹内就是训练后所得到的各种参数文件。但是每 1 000 步所保存的参数不是自动覆盖的，而是单独的文件。在后续使用的时候可以分开使用，也可以选择在所有训练都结束之后统一保存，这里之所以采用每 1 000 步进行保存的方式，主要是为了防止在训练的过程中产生突发的内存溢出或断电的情况。

如果是采用内存较小的 GPU 进行训练，则会有 GPU 显存不够的警告。不过一般情况下这影响并不大。使用 1060 显卡进行训练大概要花费 10 个小时左右的时间，并且在训练的过程中会不断地报 GPU 内存不够的警告。

另外，如果训练花了 1 个小时也没有走 100 个 step，则极有可能使用的是 CPU。这个时候我们可以通过查看 CPU 和 GPU 的使用率来确认是在 CPU 还是在 GPU 下进行训练的，本示例不建议使用 CPU 进行训练。因为 CPU 的训练周期非常长，而这个时间可能会是个无底洞（因为作者使用 CPU 用了 1 个小时训练 1 步都没有走完，所以放弃了使用 CPU 进行时间测试）。训练后会在 data 目录下产生模型的各种参数文件，但是这并不是一个 ckpt 文件。如果需要 ckpt 文件，只需要在 saver.save 方法中设置具体的 ckpt 文件保存即可。

当模型训练完成后，我们可以写一个测试文件进行测试，当然也可以直接使用预

训练模型验证，当训练结束后直接将训练好的 ckpt 文件进行相应的替换即可。由于篇幅的原因，在这里我们只简单说一下测试文件的编写过程和大体思路及一些主要的代码。

编写测试文件，主要有以下几个步骤：

- (1) 定义相关模型信息并设定需要识别的图片；
- (2) 初始化网络参数；
- (3) 目标识别及框选目标。

第一步是定义模型信息和设定所需要识别的图片，代码如下：

```
parser = argparse.ArgumentParser()
parser.add_argument('--weights', default="YOLO_small.ckpt", type=str)
parser.add_argument('--weight_dir', default='weights', type=str)
parser.add_argument('--data_dir', default="data", type=str)
parser.add_argument('--gpu', default='', type=str)
args = parser.parse_args()

os.environ['CUDA_VISIBLE_DEVICES'] = args.gpu

yolo = YOLONet(False)
weight_file = os.path.join(args.data_dir, args.weight_dir, args.weights)
detector = Detector(yolo, weight_file)

imname = 'test/dg.jpg'
detector.image_detector(imname)
```

在这里，我们将权重信息、数据信息，以及 GPU 设置信息全部加载进来，然后设置 GPU 的环境变量'CUDA_VISIBLE_DEVICES'，最后加载一个名为“dg.jpg”的文件作为识别的目标。

将目标文件传入网络之后，要做的就是初始化网络参数，即 `detector = Detector(yolo, weight_file)`这句话，我们来看下这个函数中的 `init` 定义代码。

```
def __init__(self, net, weight_file):
    self.net = net
    self.weights_file = weight_file
```



```
self.classes = cfg.CLASSES
self.num_class = len(self.classes)
self.image_size = cfg.IMAGE_SIZE
self.cell_size = cfg.CELL_SIZE
self.bboxes_per_cell = cfg.BBOXES_PER_CELL
self.threshold = cfg.THRESHOLD
self.iou_threshold = cfg.IOU_THRESHOLD
self.boundary1 = self.cell_size * self.cell_size * self.num_class
self.boundary2 = self.boundary1 + \
    self.cell_size * self.cell_size * self.bboxes_per_cell

self.sess = tf.Session()
self.sess.run(tf.global_variables_initializer())

print('Restoring weights from: ' + self.weights_file)
self.saver = tf.train.Saver()
self.saver.restore(self.sess, self.weights_file)
```

这个 init 部分的代码和 train.py 文件中定义的代码实际上基本是一样的，在这里不做详细说明。

网络的参数定义好之后，接下来的工作就是进行目标识别和目标框选了。在进行目标识别时，主要调用 YOLO_small.ckpt 模型文件，获取模型中的参数，然后进行目标检测和识别，并将识别出来的结果进行框选。最后通过 OpenCV 库将结果显示出来。在这里使用了 3 张图片进行识别，结果如图 14-11、图 14-12 和图 14-13 所示，这三张图片中，前面两张图片的识别效果非常好，可以很容易地将目标框选出来，并且识别出来对应的分类。例如图 14-11 中，识别出来的分类是两只狗，图 14-12 中识别出来的是 4 个人，但是在图 14-13 中，我们可以发现，只识别出来狗的信息，而猫的信息并未被正确地识别，这是什么原因呢？



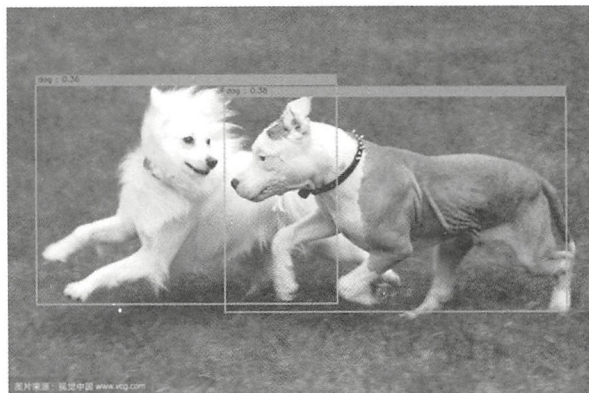


图 14-11

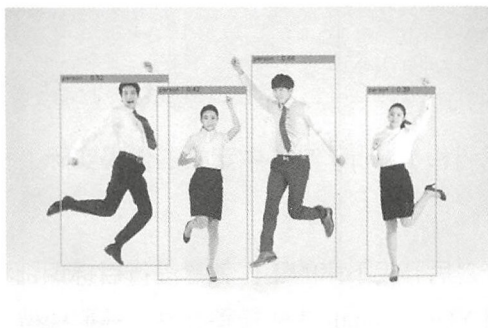


图 14-12

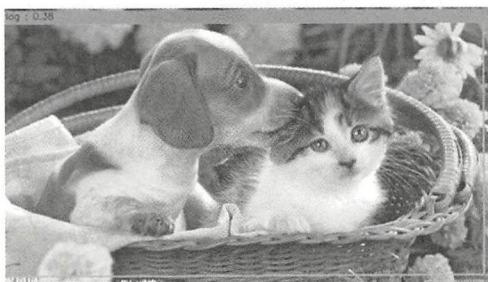


图 14-13

其实出现这种现象的原因主要有两个：第一个是因为我们目前测试所用的模型只是 YOLO 的预训练模型，而预训练模型的准确率并不是特别高；第二个原因则是 YOLO 的一个比较通用的问题，那就是针对目标相对密集的图片，识别的准确率较低，

那么为什么会出现这样的问题呢？

（1）因为在网络中只预测了 2 个框，且属于同一类，因此，YOLO 网络对于两个非常靠近的物体或非常小的群体预测效果并不好，就会产生如图 14-13 所示的结果。

（2）由于损失函数对于占比相对比较大的物体，或占比比较小的物体的处理并不好，因此在进行误差定位的时候会产生差错。

（3）对于不常见的长宽比来说，YOLO 网络不能进行很好的预测和判断，因此对于这类的图像来说效果也不是很好。



附录A

TensorFlow 历代版本更新内容

A.1 TensorFlow 1.3 版本更新内容

1.3 的编译版本是用 NVIDIA 的 cuDNN 6 编译的，而 1.2.1 用的是 cuDNN 5.1。cuDNN 新版显著提升了 Softmax 层的性能。cuDNN 6 新增的一个有趣的功能是膨胀卷积 (Dilated Convolution)，Tensorflow 已经支持此特性。需要注意的是，从 1.1.0 版本开始，Tensorflow 不再支持 Mac 上的 GPU。虽然开发者还能得到补丁，但不能保证它能正常运行。

`tf.contrib.data.Dataset` 类获得了一些重要更新。开发者可以使用这个类为自己在 Tensors 中的数据创建统一的输入流水线，输入来源可以是内存、文件或磁盘，支持多种数据格式。它还能用来对使用 `Dataset.map()` 的各个独立元素应用函数，或者对所有使用 `Dataset.batch()` 的元素应用函数。这个类中缺少嵌套结构的函数现在把列表隐式地转换成 `tf.Tensor` 对象。不想用它的用户可以使用元组来代替。`Dataset` 类中还提供了几个新函数：

- `Dataset.list_files(file_pattern)`: 返回一个 `Dataset`，包含与 `file_pattern` 模式相匹配的文件名字符串。
- `Dataset.interleave(map_func, cycle_length)`: 赋予程序员更大的自由度来处理函数到元素的映射。它仍会对整个 `Dataset` 应用 `map_func`，但会交叉结果，这样有助于



同时处理多个输入文件。

- **ConcatenateDataset**: 用于扩展 **Dataset** 类的一个类。它的 **init** 函数接收两个 **Dataset** 作为参数, 通过已有的 **Dataset.concatenate()** 函数将它们连接起来。

对于高级 API 函数和统计分布, 新增的一项是多重统计分布。使用一个类表示一个统计分布, 并用定义这个分布的参数进行初始化。现在有很多单变量和多变量的分布。开发者也能扩展已有的类, 但是必须继续支持 **Distribution** 基类中现有的函数。为避免无效的属性, 开发者可以让程序抛出一个异常, 或者选择用 **NaN** 值处理。

虽然 Keras 和 TFLearn 用户已有很多高级 API 函数可用, TensorFlow 又在库中增加了下列函数: **DNNClassifier**、**DNNRegressor**、**LinearClassifier**、**LinearRegressor**、**DNNLinearCombinedClassifier**、**DNNLinearCombinedRegressor**。它们已经成为 **tf.contrib.learn** 包的一部分。

A.2 TensorFlow 1.4 版本更新内容

在 1.4 版本中, Keras 已从 **tf.contrib.keras** 被迁移到核心软件包 **tf.keras** 中。Keras 是一个非常热门的机器学习框架, 它包含众多高级 API, 这些 API 可以大大缩短从创意到实现之间的时间。Keras 可与其他核心 TensorFlow 功能平稳集成, 包括 Estimator API。事实上, 您可以调用 **tf.keras.estimator.model_to_estimator** 函数, 直接从任何 Keras 模型构建估算器。由于 Keras 现在已添加到 TensorFlow 核心中, 您可以在生产流程中调用它。

Dataset API 已从 **tf.contrib.data** 迁移到核心软件包 **tf.data** 中。1.4 版本的 Dataset API 还增加了对 Python 生成器的支持。我们强烈建议使用 Dataset API 为 TensorFlow 模型创建输入管道, 因为:

- (1) 与旧 API (**feed_dict** 或队列式管道) 相比, Dataset API 可以提供更多功能。
- (2) Dataset API 的性能更高, 更简洁, 更易于使用。



1.4 版本还引入了实用函数 `tf.estimator.train_and_evaluate`，它简化了训练、评估和估算器模型的导出工作。此函数可以实现训练和评估的分布式执行，同时仍然支持本地执行。

A.3 TensorFlow 1.5 版本更新内容

- (1) 预构建的二进制文件针对 CUDA 9.0 和 cuDNN 7 而构建。
- (2) Linux 二进制文件是使用 Ubuntu 16 容器构建的，可能在 Ubuntu 14 中使用时会引发 glibc 不兼容问题。
- (3) TensorFlow Lite: dev 预览版已提供下载，TensorFlow Lite 是 Google I/O 2017 大会的一个重要发布，有了 TensorFlow Lite，应用开发者可以在移动设备上部署人工智能。

A.4 TensorFlow 1.6 版本更新内容

- (1) 现在预构建的二进制文件是针对 CUDA 9.0 和 cuDNN 7 构建的。
- (2) 预编译的二进制文件将使用 AVX 指令。这可能会破坏较旧的 CPU 上的 TF。
- (3) `tf.estimator.{FinalExporter, LatestExporter}` 可导出被剥离的 SavedModels，改进了 SavedModel 的向前兼容性。
- (4) FFT 支持添加到 XLA CPU/GPU。
- (5) Android TF 现在可以在兼容的 Tegra 设备上使用 CUDA 加速来构建。
- (6) 针对 ML 新手新增了第二个版本的入门指南。
- (7) 为了向后兼容，将 `prepare_variance` 布尔值默认设置为 False。



A.5 TensorFlow 1.7 版本更新内容

- (1) eager 模式正在移出 contrib, 请尝试 `tf.enable_eager_execution()`;
- (2) 图形重写模拟与 TensorFlow Lite 兼容的定点量化, 由新的 `tf.contrib.quantize` 软件包支持;
- (3) 使用 `tf.custom_gradient` 轻松定制梯度计算;
- (4) TensorBoard Debugger Plugin、TensorFlow 调试器 (`tfdbg`) 的图形用户界面 (GUI) 处于 Alpha 模式;
- (5) 使用新的 `tf.contrib.data.SqlDataset` 作为数据集读取 SQLite 数据库的实验性支持;
- (6) 分布式 Mutex / CriticalSection 添加到 `tf.contrib.framework.CriticalSection`;
- (7) 使用 `tf.regex_replace` 进行更好的文本处理;
- (8) 使用 `tf.contrib.data.bucket_by_sequence_length` 轻松有效地输入序列。

A.6 TensorFlow 1.8 版本更新内容

- (1) 可以将 `tf.contrib.distribute.MirroredStrategy()` 传递给 `tf.estimator.RunConfig()`, 能够在一台有多个 GPU 的机器上运行评估器 (Estimator) 模型。
- (2) 添加 `tf.contrib.data.prefetch_to_device()`, 支持预取 GPU 内存。
- (3) 添加梯度提升树作为预先制作的评估器: `BoostedTreesClassifier`, `BoostedTreesRegressor`。
- (4) 为云端 TPU 添加第三管道配置, 提高其性能和可用性。
- (5) `tf.contrib.bayesflow` 转向自己的 repo。



(6) 添加了 `tf.contrib.{proto, rpc}`，允许通用的原型解析和 RPC 通信。

A.7 TensorFlow 1.9 版本更新内容

(1) `tf.keras` 文件升级：新的基于 Keras 的入门以及程序员指导手册；

(2) `tf.keras` 升级到 Keras 2.1.6 API；

(3) 添加 `tf.keras.layers.CuDNNGRU` 和 `tf.keras.layers.CuDNNLSTM` 层；

(4) 对梯度提升树估算器 (Gradient Boosted Trees Estimators) 添加核心功能栏和损失 (Feature Columns and losses) 的支持；

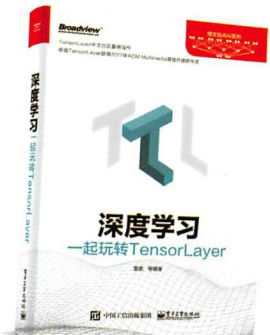
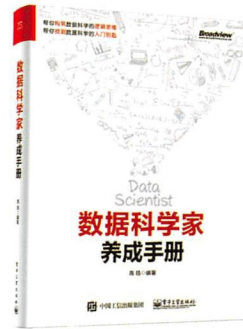
(5) TFLite 优化转换器的 Python 界面有所扩展，命令行界面 (AKA: `toco`, `tflite_convert`) 再次包含在了标准 `pip` 安装中；

(6) 优化了数据载入和文本处理；

(7) 实验性地增加了对新的预制估算器的支持：

(8) `distributions.Bijector` API 支持使用新的 API 为 `Bijectors` 广播。





本书将深度学习的理论与实践高度结合。作者以其多年的研发经验，为深度学习开发工程师们提供了一套全面系统的学习教材与实践案例。本书由浅入深地介绍了TensorFlow在各种模型和场景下的使用方案，同时也对深度学习整体框架和各种常用模型进行了详细阐述。书中配有大量的实战案例，以简单生动的语言将各种算法和结构讲解出来。对于在这个领域工作的工程师、老师、学生，这是一本难得的好书。

孙伟

北京航空航天大学软件学院创始院长，教授
北京软件行业协会 执行会长
美国佛罗里达州立大学计算机学院终身教授

人工智能时代的来临对于每个程序员来说都是机遇，也是挑战。这就好比在机械革命来临之时，一个农夫是选择继续埋头耕种，还是努力去学会制造耕地的机器一样。TensorFlow等一大批优秀的开源框架，给了每个程序员几乎平等的学习机会，让我们能够具备更高的生产能力，那我们还等什么？本书有一些有趣的应用方法、一些浅显生动的例子，相信能够给你带来一定的入门提示或者思维灵感。

高扬

珠海金山办公软件人工智能组专家

市面上关于TensorFlow机器学习的书很多，本书作者的可贵之处在于深谙机器学习的精髓，却可以用简单朴实的语言将原本艰涩难懂的框架、模型等进行了生动的解构，配合作者多年一线实战案例，使得整本书兼顾了不同阶段学习者的诉求。这份诚意应当点赞！

马志国

中科院计算机研究所博士、金山软件人工智能组算法经理

从2016年人工智能的开始火爆到现在的满地开花，人工智能领域已经给了这个社会太多的机会和机遇，无论是计算机专业毕业的同学还是数学专业毕业的同学都纷纷投入其中。本书给了开发者一个非常好的入口，从基础到实战、从理论到算法，再配合简洁生动的文字，使得这本书的可读性非常高，相信一定能给开发者们带来不一样的阅读体验。

周志明

人工智能领域专家、《深入理解Java虚拟机》作者

认识鸿波多年，他一直走在技术开发的一线，在人工智能这个领域已经有了快10年的积累，他的书也正如他的技术积累一样，朴实、睿智，书中通过简单的语言来解释较为复杂的原理，并使用TensorFlow将其实现，为TensorFlow初学者提供了非常好的技术思路，读后受益匪浅。

俞天翔

Amazon.com 高级工程师

本书以TensorFlow机器学习框架为核心主题，深入浅出地介绍了深度学习相关的理论基础、算法实现与落地应用，并以深度学习的实践应用为导向，通过目标检测、自然语言处理等实践案例，以源代码循序渐进地向读者展示技术关键点与实战经验。对于入门TensorFlow和深度学习的研发人员这是一本不可多得的好书。

陈启明

美国微软公司 高级工程师



博文视点Broadview



@博文视点Broadview

责任编辑：孙学瑛
封面设计：吴海燕

上架建议：人工智能>深度学习

ISBN 978-7-121-34565-4



定价：99.00元